

Android Database Programming

Exploit the power of data-centric and data-driven Android applications with this practical tutorial

Jason Wei



BIRMINGHAM - MUMBAI

Android Database Programming

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2012

Production Reference: 1230512

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-812-3

www.packtpub.com

Cover Image by Jason Wei (jwei512@gmail.com)

Credits

Author

Jason Wei

Project Coordinator

Joel Goveya

Reviewers

Joseph Lau

Prashant Thakkar (Pandhi)

Proofreader

Sandra Hopper

Acquisition Editor

Kartikey Pandey

Indexer

Rekha Nair

Lead Technical Editor

Azharuddin Sheikh

Graphics

Manu Joseph

Technical Editors

Ankita Shashi

Manmeet Singh Vasir

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Author

Jason Wei graduated from Stanford University in 2011 with a B.S. in Mathematical Computational Science, a minor in Statistics, and an M.S. in Management Science and Engineering with a concentration on Machine Learning. He spent his first two years in college with startups in Silicon Valley, and it was at his second startup (BillShrink, Inc) that he was introduced to Android.

Since then he has developed a handful of applications ranging from silly screen prank applications to serious financial pricing and modeling tools. He also enjoys working with APIs and competing in application development contests - winning a number of contests hosted by companies like Google, MyGengo, IndexTank, amongst others. In addition to developing applications, Jason enjoys writing Android tutorials and sharing his own development experiences on his blog (thinkandroid.wordpress.com), and it was through his blog that he was first invited to be a technical reviewer for the book *Learning Android Game Programming*.

Jason is currently working as a quantitative trader in New York.

About the Reviewers

Joseph Lau is currently a graduate student at Stanford University, studying towards his M.S. in Computer Science. During his summers, he's interned at LinkedIn and Google in various technical positions. Android programming is a hobby of his, and he has written several Android applications. He believes mobile applications are a key component of technical innovation in the 21st century and thinks it's a great time to pick up Android programming if you haven't yet.

Prashant Thakkar (Pandhi) is a Technical Lead with more than seven years of IT experience. His strengths are Java, J2EE with frameworks like Struts, Hibernate, and related open source frameworks. Prashant has been working on Android for more than two years and has delivered mission-critical Enterprise Mobile Applications. His interests also include Google App Engine for delivering applications in the cloud. Prashant writes about his technical experiments on his blogs at <http://ppandhi.wordpress.com> and <http://androidpartaker.wordpress.com>

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Storing Data on Android	7
Using SharedPreferences	8
Common use cases for SharedPreferences	10
Checking if it's the user's first time visit to your application	10
Checking when the application last updated itself	11
Remembering what the user's login username was	12
Remembering an application's state	12
Caching a user's location	12
Internal storage methods	13
External storage methods	16
SQLite databases	20
Summary	25
Chapter 2: Using a SQLite Database	27
Creating advanced SQLite schemas	27
Wrappers for your SQLite database	30
Debugging your SQLite database	40
Summary	42
Chapter 3: SQLite Queries	43
Methods for building SQLite queries	43
SELECT statements	45
WHERE filters and SQL operators	49
DISTINCT and LIMIT clauses	52
ORDER BY and GROUP BY clauses	55
HAVING filters and Aggregate functions	59
SQL vs. Java performance comparisons	66
Summary	71

Chapter 4: Using Content Providers	73
ContentProvider	73
Implementing the query method	79
Implementing the delete and update methods	82
Implementing the insert and getType methods	86
Interacting with a ContentProvider	90
Practical use cases	92
Summary	94
Chapter 5: Querying the Contacts Table	95
Structure of the Contacts content provider	95
Querying for Contacts	98
Modifying Contacts	102
Setting permissions	107
Summary	108
Chapter 6: Binding to the UI	109
SimpleCursorAdapters and ListViews	109
Custom CursorAdapters	114
BaseAdapters and Custom BaseAdapters	117
Handling list interactions	123
Comparing CursorAdapters and BaseAdapters	125
Summary	126
Chapter 7: Android Databases in Practice	129
Local database use cases	130
Databases as caches	134
Typical application design	137
Summary	139
Chapter 8: Exploring External Databases	141
Different external databases	141
Google App Engine and JDO databases	143
GAE: an example with video games	145
The PersistenceManager and Queries	148
Summary	156
Chapter 9: Collecting and Storing Data	157
Methods for collecting data	157
A primer on web scraping	159
Extending HTTP servlets for GET/POST methods	170
Scheduling CRON jobs	174
Summary	176

Chapter 10: Bringing it Together	177
Implementing HTTP GET requests	177
Back to Android: parsing responses	181
Final steps: binding to the UI (again)	187
Summary	192
Index	193

One last thing to note before we move on to typical use cases of `SharedPreferences` is that if you decide to set the visibility of your shared preference instance to `MODE_WORLD_WRITEABLE`, then you are potentially exposing yourself to various security breaches by malicious external applications. As a result, in practice, this mode is not recommended. However, the desire to share information locally between two applications is still one that many developers face, and so a method for doing so was developed that simply involves setting an `android:sharedUserId` in your application's manifest files.

How this works is that each application, when signed and exported, is given an auto-generated application ID. However, if you explicitly set this ID in the application's manifest file, then, assuming two applications are signed with the same key, they will be able to freely access each other's data without having to expose their data to the rest of the applications on a user's phone. In other words, by setting the same ID for two applications, those two *and only those two* applications will be able to access each other's data.

Common use cases for SharedPreferences

Now that we know how to instantiate and edit a shared preference object, it's important to think about some typical use cases for this type of data storage. And so, following are a couple of examples, illustrating what kinds of small, primitive key-value data pairs applications tend to like to save.

Checking if it's the user's first time visit to your application

For many applications, if this is the user's first visit, then they will want to display some kind of instructions/tutorials activity or a splash screen activity:

```
public class SharedPreferencesExample2 extends Activity {
    private static final String MY_DB = "my_db";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        SharedPreferences sp = getSharedPreferences(MY_DB,
            Context.MODE_PRIVATE);
        /**
         * CHECK IF THIS IS USER'S FIRST VISIT
         */
        boolean hasVisited = sp.getBoolean("hasVisited",
            false);
    }
}
```

```

        if (!hasVisited) {
            // ...
            // SHOW SPLASH ACTIVITY, LOGIN ACTIVITY, ETC
            // ...
            // DON'T FORGET TO COMMIT THE CHANGE!
            Editor e = sp.edit();
            e.putBoolean("hasVisited", true);
            e.commit();
        }
    }
}

```

Checking when the application last updated itself

Many applications will have some kind of caching, or syncing, feature built-in, which will require regular updating. By saving the last update time, we can quickly check to see how much time has elapsed, and decide whether or not an update/sync needs to occur:



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

```

/**
 * CHECK LAST UPDATE TIME
 */
long lastUpdateTime = sp.getLong("lastUpdateKey", 0L);
long timeElapsed = System.currentTimeMillis() -
lastUpdateTime;
// YOUR UPDATE FREQUENCY HERE
final long UPDATE_FREQ = 1000 * 60 * 60 * 24;
if (timeElapsed > UPDATE_FREQ) {
    // ...
    // PERFORM NECESSARY UPDATES
    // ...
}
// STORE LATEST UPDATE TIME
Editor e = sp.edit();
e.putLong("lastUpdateKey", System.currentTimeMillis());
e.commit();

```

Remembering what the user's login username was

Many applications will allow the user to remember their username (as well as other login-oriented fields such as PINs, phone numbers, and so on) and a shared preference is a great way to store a simple primitive string ID:

```
/**
 * CACHE USER NAME AS STRING
 */
// TYPICALLY YOU WILL HAVE AN EDIT TEXT VIEW
// WHERE THE USER ENTERS THEIR USERNAME
EditText userNameLoginText = (EditText)
    findViewById(R.id.login_editText);
String userName =
    userNameLoginText.getText().toString();
Editor e = sp.edit();
e.putString("userNameCache", userName);
e.commit();
```

Remembering an application's state

For many applications, the functionality of the application will change depending on the application's state, typically set by the user. Consider a phone ringer application – if the user specifies that no functionality should occur if the phone is in silent mode, then this is probably an important state to remember:

```
/**
 * REMEMBERING A CERTAIN STATE
 */
boolean isSilentMode = sp.getBoolean("isSilentRinger",
    false);
if (isSilentMode) {
    // ...
    // TURN OFF APPLICATION
    // ...
}
```

Caching a user's location

Any location-based application will often want to cache the user's last location for a number of reasons (perhaps the user has turned off GPS, or has a weak signal, and so on). This can be easily done by converting the latitude and longitude of the user to floats and then storing those floats in a shared preference instance:

```
/**
 * CACHING A LOCATION
```

```

*/
// INSTANTIATE LOCATION MANAGER
LocationManager locationManager = (LocationManager)
    this.getSystemService(Context.LOCATION_SERVICE);
// ...
// IGNORE LOCATION LISTENERS FOR NOW
// ...
Location lastKnownLocation =
    locationManager.getLastKnownLocation
        (LocationManager.NETWORK_PROVIDER);
float lat = (float) lastKnownLocation.getLatitude();
float lon = (float) lastKnownLocation.getLongitude();
Editor e = sp.edit();
e.putFloat("latitudeCache", lat);
e.putFloat("longitudeCache", lon);
e.commit();

```

With the latest version of Android (API Level 11), there is also a new `getStringSet()` method which allows you to set and retrieve a set of string objects for a given associated key. Here's how it looks in action:

```

Set<String> values = new HashSet<String>();
values.add("Hello");
values.add("World");
Editor e = sp.edit();
e.putStringSet("strSetKey", values);
e.commit();
Set<String> ret = sp.getStringSet(values, new HashSet<String>());
for(String r : ret) {
    Log.i("SharedPreferencesExample", "Retrieved vals: " + r);
}

```

Use cases for this are plenty — but for now let's move on.

Internal storage methods

Let's begin with internal storage mechanisms on Android. For those with experience in standard Java programming, this section will come pretty naturally. Internal storage on Android simply allows you to read and write to files that are associated with each application's internal memory. These files can only be accessed by the application and cannot be accessed by other applications or by the user. Furthermore, when the application is uninstalled, these files are automatically removed as well.

The following is a simple example of how to access an application's internal storage:

```
public class InternalStorageExample extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // THE NAME OF THE FILE
        String fileName = "my_file.txt";
        // STRING TO BE WRITTEN TO FILE
        String msg = "Hello World.";
        try {
            // CREATE THE FILE AND WRITE
            FileOutputStream fos = openFileOutput(fileName,
                Context.MODE_PRIVATE);
            fos.write(msg.getBytes());
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Here we simply use the `Context` class's `openFileOutput()` method, which takes as its first argument the name of the file to be created (or overridden) and as its second argument the visibility of that file (just like with `SharedPreferences`, you can control the visibility of your files). It then converts the string we want to write to byte form and passes it into the output stream's `write()` method. One thing to mention though is an additional mode that can be specified with `openFileOutput()` and that is:

- `MODE_APPEND`: This mode allows you to open an existing file and append a string to its existing contents (any other mode and the existing contents will be deleted)

Furthermore, if you are programming in Eclipse, then you can go to the **DDMS** screen and look at your application's internal files (amongst other things):

File Explorer						
Name	Size	Date	Time	Permissions	Info	
▶		2009-12-29	09:01	drwxr-xr-x		
▶		2009-12-29	09:01	drwxr-xr-x		
▶		2009-12-29	09:01	drwxr-xr-x		
▶		2009-12-29	09:01	drwxr-xr-x		
▶		2009-12-29	09:01	drwxr-xr-x		
▶		2010-07-04	02:43	drwxr-xr-x		
▶		2010-07-14	23:31	drwxr-xr-x		
▶		2010-01-27	11:35	drwxr-xr-x		
▶		2010-01-27	11:43	drwxr-xr-x		
▶		2009-12-31	04:19	drwxr-xr-x		
▶		2009-12-31	21:08	drwxr-xr-x		
▶		2010-07-10	19:25	drwxr-xr-x		
▶		2009-12-29	09:01	drwxr-xr-x		
▶		2012-02-04	03:56	drwxr-xr-x		
▶		2012-03-27	02:59	drwxrwx--x		
▶		2012-03-27	02:59	-rw-rw----		
▶		2012-02-04	03:56	drwxr-xr-x		
▶		2009-12-29	09:00	drwxr-x---		
▶		2009-12-29	09:00	drwxrwx--x		
▶		2009-12-29	09:00	drwxrwx---		
▶		2009-12-29	09:00	drwxrwx--t		
▶		2009-12-29	09:00	drwx-----		
▶		2009-12-29	09:01	drwxrwxr-x		
▶		2012-03-27	02:55	d-----		
▶		2009-10-22	08:33	drwxr-xr-x		

And so we see the text file that we just created. For those developing with the terminal, the path for this would be `/data/data/{your-app-path}/files/my_file.txt`. Now, unfortunately, reading back files is much more verbose and the code for how you would do that looks like:

```
public class InternalStorageExample2 extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        // THE NAME OF THE FILE
        String fileName = "my_file.txt";
        try {
            // OPEN FILE INPUT STREAM THIS TIME
            FileInputStream fis = openFileInput(fileName);
            InputStreamReader isr = new InputStreamReader(fis);
            // READ STRING OF UNKNOWN LENGTH
            StringBuilder sb = new StringBuilder();
            char[] inputBuffer = new char[2048];
            int l;
            // FILL BUFFER WITH DATA
            while ((l = isr.read(inputBuffer)) != -1) {
```



```
        sb.append(inputBuffer, 0, 1);
    }
    // CONVERT BYTES TO STRING
    String readString = sb.toString();
    Log.i("LOG_TAG", "Read string: " + readString);
    // CAN ALSO DELETE THE FILE
    deleteFile(fileName);
} catch (IOException e) {
    e.printStackTrace();
}
}
```

Here we start by opening a file input stream instead and pass it into a stream reader. This will allow us to call the `read()` method and read in the data as bytes which we can then append to a `StringBuilder`. Once the contents have been read back fully, we simply return the `String` from the `StringBuilder` and voila! At the end, just for the sake of completeness, the `Context` class provides you with a simple method for deleting files saved in the internal storage.

External storage methods

External storage, on the other hand, involves storing data and files to the phone's external **Secure Digital (SD)** card. The concept behind internal and external storage is similar, and so let's begin by laying down the pros and cons of this kind of storage versus what we saw earlier — that is, `SharedPreferences`. In a shared preference, there is much less overhead and so reading/writing to a simple `Map` object is much more efficient than reading/writing to a disk. However, because you are limited to simple primitive values (for the most part; again the most recent version of Android allows you to save sets of strings), you are essentially trading flexibility for efficiency. With internal and external storage mechanisms, you can save not only much bigger chunks of data (that is, entire XML files) but also much more complicated forms of data (that is, media files, image files, and so on).

Now, how about internal versus external storage? Well the pros and cons of these two are much more subtle. First, let's consider the amount of **storage space** (*memory*). Though this varies depending on the phone a user owns, the amount of internal memory can often be quite low, and it is not uncommon for even relatively new phones to have as low as 512 MB of internal storage. External storage, on the other hand, depends solely on what SD card the user has in their phone. Typically, if an SD card is present, then the amount of external storage can be many times greater than the amount of internal storage (depending on the size of the SD card, this can be as high as 32 GB of storage).

Now, let's consider the **access speed** for internal versus external storage.

Unfortunately, in this case, nothing conclusive can be drawn as the read and write speeds are highly dependent on the type of internal flash memory the phone uses, as well as the classification of the SD card for external storage. And so the last thing to consider is the *accessibility* of each type of storage mechanism. Again, for internal storage, the data is only accessible by your application, and so it is extremely safe from potentially malicious external applications. The con is that if the application is uninstalled, then that internal memory is wiped as well. For external storage, the visibility is inherently world readable and writeable, and so any files saved are exposed both to external applications as well as to the user. There is no guarantee then that your files will remain safe and uncorrupted.

Now that we've flushed out some of the differences, let's get back to the code and see how you can actually access the external SD card with this following example:

```
public class ExternalStorageExample extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        String fileName = "my_file.txt";
        String msg = "Hello World.";
        boolean externalAvailable = false;
        boolean externalWriteable = false;
        String state = Environment.getExternalStorageState();
        if (state.equals(Environment.MEDIA_MOUNTED)) {
            // HERE MEDIA IS BOTH AVAILABLE AND WRITEABLE
            externalAvailable = true;
            externalWriteable = true;
        } else if
            (state.equals(Environment.MEDIA_MOUNTED_READ_ONLY)) {
            // HERE SD CARD IS AVAILABLE BUT NOT WRITEABLE
            externalAvailable = true;
        } else {
            // HERE FAILURE COULD BE RESULT OF MANY SITUATIONS
            // NO OP
        }
        if (externalAvailable && externalWriteable) {
            // FOR API LEVEL 7 AND BELOW
            // RETRIEVE SD CARD DIRECTORY
            File r = Environment.getExternalStorageDirectory();
            File f = new File(r, fileName);
            try {
```

```
        // NOTE DIFFERENT FROM INTERNAL STORAGE WRITER
        FileWriter fWriter = new FileWriter(f);
        BufferedWriter out = new BufferedWriter(fWriter);
        out.write(msg);
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
} else {
    Log.e("LOG_TAG", "SD CARD UNAVAILABLE");
}
}
```

In order to execute the previous code, don't forget to add into your manifest file the `WRITE_EXTERNAL_STORAGE` permission. Here, we start by making a call to the `Environment` class's `getExternalStorageState()` method, which allows us to detect whether an external SD card is actually mounted and writeable. Trying to read or write to a file without performing these preliminary checks will cause an error to be thrown.

Once we know that an SD card is mounted and, indeed, writeable, then for those with API Levels 7 and below, we call `getExternalStorageDirectory()` to retrieve the file path to the root of the SD card. At this point, we simply need to create our new file and instantiate a `FileWriter` and `BufferedWriter` and write our string to the file. One thing to note here is that the method for writing to disk when dealing with external storage differs from our previous method for writing to disk with internal storage.

This is actually an important point to note and understand, which is why I place so much emphasis on these write methods. In the internal storage example, we obtained a `FileOutputStream` object by calling the `Context` class's `openFileOutput()` method, which took as its second argument a mode. When passing in `MODE_PRIVATE`, what happens behind the scenes is that each time a file is created and written to with that `FileOutputStream`, that file is encrypted and signed with your application's unique ID (as mentioned earlier), so that external applications cannot access the contents of those files. However, remember that when creating and writing to files in *external storage*, by default they are created with no security enforcements, so any application (or user) can both read and write to those files. This is why you can use standard Java methods (for example, `FileWriter`) for writing to external SD cards, but not when writing to internal storage. One last thing to note is that just as you can see the newly created file in the **DDMS** perspective in Eclipse, assuming you have an SD card setup, you can just as easily see the newly created text file in **DDMS**:

Name	Size	Date	Time	Permissions	Info
data		2009-12-25	06:43	drwxrwx--x	
anr		2009-12-25	06:44	drwxrwxrwx	
app		2009-12-25	06:43	drwxrwx--x	
app-private		2009-12-25	06:43	drwxrwx--x	
dalvik-cache		2009-12-25	06:43	drwxrwx--x	
data		2009-12-25	06:43	drwxrwx--x	
local		2009-12-25	06:43	drwxrwx--x	
lost+found		2009-12-25	06:43	drwxrwx---	
misc		2009-12-25	06:43	drwxrwx--t	
property		2009-12-25	06:43	drwx-----	
system		2009-12-25	06:44	drwxrwxr-x	
tombstones		2011-05-13	04:30	drwxr-xr-x	
sdcard		1970-01-01	00:00	d---rwxrwx	
my_file.txt	12	2012-03-27	03:05	----rw-rw-	
system		2009-04-22	04:11	drwxr-xr-x	

So while developing your application, by leveraging this **DDMS** perspective you can quickly push, pull, and monitor files that you are writing to disk.

With that said, I'll quickly mention some of the changes in writing to external storage that were introduced after API Level 8. These changes are actually very well documented at [http://developer.android.com/reference/android/content/Context.html#getExternalFilesDir\(java.lang.String\)](http://developer.android.com/reference/android/content/Context.html#getExternalFilesDir(java.lang.String))

But from a high level, in API Level 8 and above, we simply have two new primary methods:

```
getExternalFilesDir(String type)
getExternalStoragePublicDirectory(String type)
```

You'll notice that for each of these methods you can now pass in a type parameter. These type parameters allow you to specify what kind of file yours is, so that it gets organized into the right subfolders. In the first method, the external file directory root that is returned is specific to your application, so that when your application is uninstalled all of those associated files are deleted from the external SD card as well. In the second method, the file directory root that is returned is a public one, so that files stored on these paths will remain persistent even when your application is uninstalled. Deciding which to use simply depends on the kind of file you are trying to save — for instance, if it's a media file that gets played in your application, then the user probably has no use for it if he/she decides to uninstall your application.

However, say your application allows the user to download wallpapers for their phone: in this case, you might consider saving any image files to a public directory, so that even if the user uninstalls your application, those files will still be accessible by the system. The different `type` parameters that you can specify are:

```
DIRECTORY_ALARMS
DIRECTORY_DCIM
DIRECTORY_DOWNLOADS
DIRECTORY_MOVIES
DIRECTORY_MUSIC
DIRECTORY_NOTIFICATIONS
DIRECTORY_PICTURES
DIRECTORY_PODCASTS
DIRECTORY_RINGTONES
```

And so we wrap up our somewhat lengthy discussion on internal and external storage mechanisms and dive right into the even heftier topic of SQLite databases.

SQLite databases

Last, but not least, by far the most sophisticated and, arguably, the most powerful method for local storage is with SQLite databases. Each application is equipped with its own SQLite database, which is accessible by any class in the application, but not by any outside applications. Before moving on to complex queries or snippets of code, let me just give a quick summary of what SQLite databases are.

SQL (Structured Query Language) is a programming language designed especially for managing data in *relational* databases. **Relational databases** allow you to submit insert, delete, update, and get queries, while also allowing you to create and modify schemas (more simply thought of as tables). **SQLite** then is simply a scaled-down version of MySQL, PostgreSQL, and other popular database systems. It is entirely self-contained and server-less, while still being transactional and still using the standard SQL language for executing queries. Because of how it's self-contained and executable, it is extremely efficient, flexible, and accessible by a wide variety of programming languages across a wide variety of platforms (including our very own Android platform).

For now, let's simply take a look at how we would instantiate a new SQLite database schema and create a very simple table with this code snippet:

```
public class SQLiteHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "my_database.db";
    // TOGGLE THIS NUMBER FOR UPDATING TABLES AND DATABASE
```

```

private static final int DATABASE_VERSION = 1;
// NAME OF TABLE YOU WISH TO CREATE
public static final String TABLE_NAME = "my_table";
// SOME SAMPLE FIELDS
public static final String UID = "_id";
public static final String NAME = "name";
SQLiteHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL("CREATE TABLE " + TABLE_NAME + " (" + UID + "
        INTEGER PRIMARY KEY AUTOINCREMENT," + NAME
        + " VARCHAR(255));");
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
    int newVersion) {
    Log.w("LOG_TAG", "Upgrading database from version " +
        oldVersion + " to " + newVersion + ",
        which will destroy all old data");
    // KILL PREVIOUS TABLE IF UPGRADED
    db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
    // CREATE NEW INSTANCE OF TABLE
    onCreate(db);
}
}

```

Here, the first thing we'll notice is that in order to create a customizable database schema, we must override the `SQLiteOpenHelper` class. By overriding it, we can then override the `onCreate()` method, which will allow us to dictate the structure of the table. In our case, you'll notice that we're simply creating a table with two columns, an ID column and a name column. The query is equivalent to running the following command in SQL:

```

CREATE TABLE my_table (_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(255));

```

You'll also see that the ID column has been designated as a `PRIMARY KEY` and given the `AUTOINCREMENT` property – this is actually recommended for all tables created in Android and we'll adhere to this standard going forward. Lastly, you'll see that the name column was declared a string type with maximum character length of 255 (for longer strings, we can simply type the column as a `LONGTEXT` type).

After overriding the `onCreate()` method, we also override the `onUpgrade()` method. This allows us to quickly and simply change the structure of our table. All you need to do is increment the `DATABASE_VERSION` integer and the next time you instantiate the `SQLiteHelper`, it will automatically call its `onUpgrade()` method, at which point we will first drop the old version of the database and then create the new version.

Finally, let's take a quick look at how we would insert and query for values in our very basic, bare-bones table:

```
public class SQLiteExample extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // INIT OUR SQLITE HELPER
        SQLiteHelper sqh = new SQLiteHelper(this);

        // RETRIEVE A READABLE AND WRITEABLE DATABASE
        SQLiteDatabase sqdb = sqh.getWritableDatabase();

        // METHOD #1: INSERT USING CONTENTVALUE CLASS
        ContentValues cv = new ContentValues();
        cv.put(SQLiteHelper.NAME, "jason wei");

        // CALL INSERT METHOD
        sqdb.insert(SQLiteHelper.TABLE_NAME, SQLiteHelper.NAME,
            cv);

        // METHOD #2: INSERT USING SQL QUERY
        String insertQuery = "INSERT INTO " +
            SQLiteHelper.TABLE_NAME +
            " (" + SQLiteHelper.NAME + ") VALUES ('jwei')";
        sqdb.execSQL(insertQuery);

        // METHOD #1: QUERY USING WRAPPER METHOD
        Cursor c = sqdb.query(SQLiteHelper.TABLE_NAME,
            new String[] { SQLiteHelper.UID, SQLiteHelper.NAME },
            null, null, null, null, null);
```

```

while (c.moveToNext()) {
    // GET COLUMN INDICES + VALUES OF THOSE COLUMNS
    int id = c.getInt(c.getColumnIndex(SQLiteHelper.UID));
    String name =
        c.getString(c.getColumnIndex(SQLiteHelper.NAME));
    Log.i("LOG_TAG", "ROW " + id + " HAS NAME " + name);
}

c.close();

// METHOD #2: QUERY USING SQL SELECT QUERY
String query = "SELECT " + SQLiteHelper.UID + ", " +
    SQLiteHelper.NAME + " FROM " + SQLiteHelper.TABLE_NAME;
Cursor c2 = sqdb.rawQuery(query, null);

while (c2.moveToNext()) {
    int id =
        c2.getInt(c2.getColumnIndex(SQLiteHelper.UID));
    String name =
        c2.getString(c2.getColumnIndex(SQLiteHelper.NAME));
    Log.i("LOG_TAG", "ROW " + id + " HAS NAME " + name);
}
c2.close();

// CLOSE DATABASE CONNECTIONS
sqdb.close();
sqh.close();
}
}

```

Pay close attention to this example, as it will set the path for the next couple of chapters. In this example, we first instantiate our `SQLiteHelper` and obtain a writeable `SQLiteDatabase` object. We then introduce the `ContentValues` class, which is a very convenient wrapper method that allows you to quickly insert, update, or remove rows in your table. Here you'll notice that since our ID column was created with the `AUTOINCREMENT` field, we don't need to manually assign or increment our IDs when inserting rows. Thus, we only need to pass to the `ContentValues` object the non-ID fields: in our case just the name column.

Afterwards, we go back to our `SQLiteDatabase` object and call its `insert()` method. The first argument is simply the name of the database, and the third argument is the `ContentValue` we just created. The second argument is the only *tricky* one — basically, in the event that an empty `ContentValue` is passed in, because a `SQLite` database cannot insert an empty row, whatever column is passed in as the second argument, the `SQLite` database will automatically set the value of that column to `null`. By doing so, we can better avoid `SQLite` exceptions from being thrown.

Additionally, we can insert rows into our database by just passing in a raw SQL query, as shown in the second method, to the `execSQL()` method. Lastly, now that we've inserted two rows into our table, let's practice getting and reading these rows back. Here I show two methods as well — the first is by using the `SQLiteDatabase` helper method `query()`, and the second is by executing a raw SQL query. In both cases, a `Cursor` object is returned, which you can think of as an iterator over the rows of the sub-table that is returned by your query:

```
while (c.moveToNext()) {
    // GET COLUMN INDICES + VALUES OF THOSE COLUMNS
    int id = c.getInt(c.getColumnIndex(SQLiteHelper.UID));
    String name = c.getString(c.getColumnIndex(SQLiteHelper.NAME));
    Log.i("LOG_TAG", "ROW " + id + " HAS NAME " + name);
}
```

Once we have the desired `Cursor`, the rest is straightforward. Because the `Cursor` behaves like an iterator, in order to retrieve each row we need to throw it into a `while` loop, and in each loop, we move the cursor down one row. Then, within the `while` loop we get the column indices of the columns we want to pull data from: in our case, let's just get both columns, though in practice often times you'll only want data from specific columns at any given time. Finally, pass these column indices into the proper `get()` methods of `Cursor` — namely, if the type of the column is an integer, then call the `getInt()` method; if it is a string, then call the `getString()` method, and so on.

But again, what we see here are simply the building blocks leading up to a wealth of tools and weapons that will soon be at our disposal. Soon we'll look at how we can write various wrapper methods to simplify our lives when developing large-scale applications, as well as dig further into the various methods and parameters the `SQLiteDatabase` class provides us with.

Summary

In this first chapter, we accomplished a lot. We started off by looking at the simplest and most efficient data storage method of them all — the `SharedPreferences` class. We looked at the pros and cons of using a `SharedPreferences` object in your application, and though the class itself is limited to storing primitive data types, we saw that its use cases are plenty.

Then, we moved up a little in complexity and examined both internal and external storage mechanisms. Though not as intuitive and efficient as a shared preference object, by leveraging internal and external storage, we are capable of storing both much more data and much more complex data (that is, images, media files, and so on). The pros and cons of using internal storage versus external storage are much more subtle and many times are highly phone and hardware dependent. But in any case, this goes to illustrate my earlier point that part of mastering data on Android is being able to analyze the pros and cons of every storage method and intelligently decide the most suitable method for your application's needs.

Finally, we dipped our toes into SQLite databases and looked at how you can override the `SQLiteOpenHelper` class to create your custom SQLite database and table. From there we saw an example of how to open and retrieve this SQLite database from an `Activity` class, and subsequently, how to both insert into and retrieve rows from our table. Because of the flexibility of the `SQLiteDatabase` class, we saw that there were multiple ways for both inserting and retrieving data, allowing those less familiar with SQL to utilize the wrapper methods, while allowing those SQL aficionados to flex their querying prowess by executing raw SQL commands.

In the next chapter, we'll focus on SQLite databases, and attempt to build a much more complex, yet realistic, database schema.

2

Using a SQLite Database

Earlier we were introduced to various methods for storing data on Android – data ranging from small and simple primitive values to large and complex file types. In this chapter, we'll dive deeper into an extremely powerful and efficient way to save and retrieve structured data: namely, by using SQLite databases. For the time being, we'll focus on the versatility and robustness of the SQLite database as a local backend for your application, before switching focus in later chapters and looking at ways to bind this SQLite backend with the user interface frontend.

Creating advanced SQLite schemas

In the previous chapter, we ran through a simple example of creating and using a table with two fields: an integer ID field and a String name field. However, oftentimes the database schema that your application will need will require much more than one table. And so, now that you suddenly need multiple tables, some potentially dependent on one another, how can you effectively leverage the `SQLiteOpenHelper` class to make the development of the application clean and straightforward without compromising the robustness of your schema? Let's walk through an example together to tackle this problem!

Consider a simple schema with three tables: the first a `Students` table with fields ID, name, state, and grade, and the second a `Courses` table with fields ID, and name, and the third a `Classes` table with fields ID, student ID, and course ID. What we're going to try and create is a schema where we can add/remove students, add/remove courses, and enroll/drop students from different courses. Some of the challenges we can immediately think of are as follows:

- How do we obtain simple analytics, such as number of students per course?
- What happens when we drop a course with students in it?
- What happens when we remove a student who is enrolled in courses?

On that note, let's go straight into the code. We begin by defining the schema with a couple of classes:

```
public class StudentTable {

    // EACH STUDENT HAS UNIQUE ID
    public static final String ID = "_id";

    // NAME OF THE STUDENT
    public static final String NAME = "student_name";

    // STATE OF STUDENT'S RESIDENCE
    public static final String STATE = "state";

    // GRADE IN SCHOOL OF STUDENT
    public static final String GRADE = "grade";

    // NAME OF THE TABLE
    public static final String TABLE_NAME = "students";

}

public class CourseTable {

    // UNIQUE ID OF THE COURSE
    public static final String ID = "_id";

    // NAME OF THE COURSE
    public static final String NAME = "course_name";

    // NAME OF THE TABLE
    public static final String TABLE_NAME = "courses";

}

// THIS ESSENTIALLY REPRESENTS A MAPPING FROM STUDENTS TO COURSES
public class ClassTable {

    // UNIQUE ID OF EACH ROW - NO REAL MEANING HERE
    public static final String ID = "_id";

    // THE ID OF THE STUDENT
    public static final String STUDENT_ID = "student_id";

    // THE ID OF ASSOCIATED COURSE
    public static final String COURSE_ID = "course_id";

    // THE NAME OF THE TABLE
    public static final String TABLE_NAME = "classes";

}
```

And here's the code for creating the database schema (this should look very similar to what we saw earlier):

```
public class SchemaHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "adv_data.db";

    // TOGGLE THIS NUMBER FOR UPDATING TABLES AND DATABASE
    private static final int DATABASE_VERSION = 1;

    SchemaHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // CREATE STUDENTS TABLE
        db.execSQL("CREATE TABLE " + StudentTable.TABLE_NAME
            + " (" + StudentTable.ID + " INTEGER PRIMARY KEY
            AUTOINCREMENT, "
            + StudentTable.NAME + " TEXT, "
            + StudentTable.STATE + " TEXT, "
            + StudentTable.GRADE + " INTEGER);");

        // CREATE COURSES TABLE
        db.execSQL("CREATE TABLE " + CourseTable.TABLE_NAME +
            " (" + CourseTable.ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
            + CourseTable.NAME + " TEXT);");

        // CREATE CLASSES MAPPING TABLE
        db.execSQL("CREATE TABLE " + ClassTable.TABLE_NAME +
            " (" + ClassTable.ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
            + ClassTable.STUDENT_ID + " INTEGER, "
            + ClassTable.COURSE_ID + " INTEGER);");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion,
        int newVersion) {
        Log.w("LOG_TAG", "Upgrading database from version "
            + oldVersion + " to " + newVersion + ",
            which will destroy all old data");

        // KILL PREVIOUS TABLES IF UPGRADED
        db.execSQL("DROP TABLE IF EXISTS " + StudentTable.TABLE_NAME);
        db.execSQL("DROP TABLE IF EXISTS " + CourseTable.TABLE_NAME);
        db.execSQL("DROP TABLE IF EXISTS " + ClassTable.TABLE_NAME);

        // CREATE NEW INSTANCE OF SCHEMA
        onCreate(db);
    }
}
```

So here we see that in our `onCreate()` method we execute SQL commands to create all three tables, and furthermore, in the `onUpgrade()` method we execute SQL commands that drop all three tables and subsequently recreate all three tables. Of course, since we are overriding the `SQLiteOpenHelper` class, in theory we can customize the behavior of these methods in any way we want (for instance, some developer's might not want to drop the entire schema in the `onUpgrade()` method), but for now let's keep the functionality simple.

At this point, for those who are well versed in SQL programming and database schemas, you might be wondering if it's possible to add triggers and key constraints to your SQLite database schemas. The answer is, "yes, you can use triggers but no, you cannot use foreign key constraints." In any case, to spend any time on writing and implementing triggers would be deviating too much from the core content of this book, and so I chose to omit that discussion (though these could certainly be helpful even in our simple example).

Now that we have our schema created, before moving on to designing all kinds of complex queries for pulling different groups of data (this we'll see in the next chapter), it's time to write some wrapper methods. This will help us to address some of the questions mentioned previously, which will ultimately help us in creating a robust database.

Wrappers for your SQLite database

So we have this somewhat complicated schema in front of us now, and earlier we mentioned the questions of what would happen if we removed a student who is enrolled in courses, and vice versa what would happen if we dropped a course with multiple students enrolled in it? Certainly, we wouldn't want either case to happen – in the first, we'd have courses filled with students who are no longer even enrolled in the university, and in the second, we'd have students showing up for courses that are no longer even being offered!

And so it's time to enforce some of these rules and we'll do this by adding some convenient methods to our `SchemaHelper` class. Again, some of these rules could be enforced by using trigger statements (remember that Android's SQLite database doesn't support key constraints), but one of the benefits of using wrapper methods is that they are much more intuitive to developers who may be new to your application's code base. By using a wrapper class, a developer is able to safely interact with a database whose schema the developer may know very little about. Now, let's start by tackling the simple wrappers:

```
public class SchemaHelper extends SQLiteOpenHelper {  
    private static final String DATABASE_NAME = "adv_data.db";
```

```

// TOGGLE THIS NUMBER FOR UPDATING TABLES AND DATABASE
private static final int DATABASE_VERSION = 1;

SchemaHelper(Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
    ...
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion) {
    ...
}

// WRAPPER METHOD FOR ADDING A STUDENT
public long addStudent(String name, String state, int grade) {
    // CREATE A CONTENTVALUE OBJECT
    ContentValues cv = new ContentValues();
    cv.put(StudentTable.NAME, name);
    cv.put(StudentTable.STATE, state);
    cv.put(StudentTable.GRADE, grade);

    // RETRIEVE WRITEABLE DATABASE AND INSERT
    SQLiteDatabase sd = getWritableDatabase();
    long result = sd.insert(StudentTable.TABLE_NAME,
StudentTable.NAME, cv);
    return result;
}

// WRAPPER METHOD FOR ADDING A COURSE
public long addCourse(String name) {
    ContentValues cv = new ContentValues();
    cv.put(CourseTable.NAME, name);

    SQLiteDatabase sd = getWritableDatabase();
    long result = sd.insert(CourseTable.TABLE_NAME,
CourseTable.NAME, cv);
    return result;
}

// WRAPPER METHOD FOR ENROLLING A STUDENT INTO A COURSE
public boolean enrollStudentClass(int studentId, int courseId) {
    ContentValues cv = new ContentValues();

```



```
        cv.put(ClassTable.STUDENT_ID, studentId);
        cv.put(ClassTable.COURSE_ID, courseId);

        SQLiteDatabase sd = getWritableDatabase();
        long result = sd.insert(ClassTable.TABLE_NAME,
            ClassTable.STUDENT_ID, cv);
        return (result >= 0);
    }
}
```

Now we have three simple wrapper methods for adding data into our schema. The first two involve adding new students and new courses into the database and the last one involves adding a new mapping between a student (represented by his/her ID) and a course (essentially, we are enrolling a student into a course through this mapping). Notice that in each wrapper method, we're simply adding the values into a ContentValues object, retrieving the writable SQLite database, and then inserting this ContentValues as a new row into the specified table. Next, let's write some wrapper methods for retrieving data:

```
public class SchemaHelper extends SQLiteOpenHelper {

    public long addStudent(String name, String state, int grade) {
    }

    public long addCourse(String name) {
    }

    public boolean enrollStudentClass(int studentId, int courseId) {
    }

    // GET ALL STUDENTS IN A COURSE
    public Cursor getStudentsForCourse(int courseId) {
        SQLiteDatabase sd = getWritableDatabase();

        // WE ONLY NEED TO RETURN STUDENT IDS
        String[] cols = new String[] { ClassTable.STUDENT_ID };

        String[] selectionArgs = new String[] {
            String.valueOf(courseId) };

        // QUERY CLASS MAP FOR STUDENTS IN COURSE
        Cursor c = sd.query(ClassTable.TABLE_NAME, cols,
            ClassTable.COURSE_ID + "= ?", selectionArgs, null,
            null, null);

        return c;
    }
}
```

```

// GET ALL COURSES FOR A GIVEN STUDENT
public Cursor getCoursesForStudent(int studentId) {
    SQLiteDatabase sd = getWritableDatabase();

    // WE ONLY NEED TO RETURN COURSE IDS
    String[] cols = new String[] { ClassTable.COURSE_ID };

    String[] selectionArgs = new String[] {
        String.valueOf(studentId) };

    Cursor c = sd.query(ClassTable.TABLE_NAME, cols,
        ClassTable.STUDENT_ID + "= ?", selectionArgs, null,
        null, null);

    return c;
}

public Set<Integer> getStudentsByGradeForCourse(int courseId,
int grade) {
    SQLiteDatabase sd = getWritableDatabase();

    // WE ONLY NEED TO RETURN COURSE IDS
    String[] cols = new String[] { ClassTable.STUDENT_ID };

    String[] selectionArgs = new String[] {
        String.valueOf(courseId) };

    // QUERY CLASS MAP FOR STUDENTS IN COURSE
    Cursor c = sd.query(ClassTable.TABLE_NAME, cols,
        ClassTable.COURSE_ID + "= ?", selectionArgs, null,
        null, null);

    Set<Integer> returnIds = new HashSet<Integer>();
    while (c.moveToNext()) {
        int id = c.getInt(c.getColumnIndex
            (ClassTable.STUDENT_ID));
        returnIds.add(id);
    }

    // MAKE SECOND QUERY
    cols = new String[] { StudentTable.ID };
    selectionArgs = new String[] { String.valueOf(grade) };

    c = sd.query(StudentTable.TABLE_NAME, columns,
        StudentTable.GRADE + "= ?", selectionArgs, null, null, null);
    Set<Integer> gradeIds = new HashSet<Integer>();
    while (c.moveToNext()) {
        int id = c.getInt(c.getColumnIndex(StudentTable.ID));
        gradeIds.add(id);
    }
}

```

```
        // RETURN INTERSECTION OF ID SETS
        returnIds.retainAll(gradeIds);

        return returnIds;
    }
}
```

Here we have three fairly similar methods which allow us to get very practical datasets from our schema:

- Being able to grab a list of students in a given course
- Being able to grab a list of courses for a given student
- Lastly (just to add some complexity), being able to grab a list of students of a certain grade for a given course

Note that in all three methods we begin to play with some of the parameters in the `SQLiteDatabase` object's `query()` method, and so now seems like a great time to take a closer look at what those parameters are and what exactly we did previously:

```
public Cursor query(String table, String[] columns, String selection,
String[] selectionArgs, String groupBy, String having, String orderBy)
```

And alternatively:

```
public Cursor query(String table, String[] columns, String selection,
String[] selectionArgs, String groupBy, String having, String orderBy,
String limit)
public Cursor query(boolean distinct, String table, String[] columns,
String selection, String[] selectionArgs, String groupBy, String
having, String orderBy, String limit)
```

And just for simplicity, here's how we're calling the previous method:

```
Cursor c = sd.query(ClassTable.TABLE_NAME, cols, ClassTable.COURSE_ID
+ "= ?", selectionArgs, null, null, null);
```

So a quick explanation of the three methods. The first `query()` method is the standard one, where you specify the table in the first argument and then which columns you want to return in the second argument. This is equivalent to performing a `SELECT` statement in standard SQL. Then, in the third argument we begin to filter our query and the syntax for these filters is equivalent to including a `WHERE` clause at the end of our `SELECT` query. In our example, we see that we only ask to return the column containing student IDs, as this is the only column we care about (since we're filtering on the course ID column, it would be unnecessarily redundant to return this column as well). Then, in the filter parameter, we ask to filter by the course ID and the syntax is equivalent to passing in the following String:

```
WHERE course_id = ?
```

Here, the question mark acts as a place card for whatever value we will pass into the filter. In other words, the format of the `WHERE` statement is there, but we just need to substitute into the question mark the actual value we want to filter by. In this case, we pass into the fourth parameter the given course ID.

The last three arguments (`groupBy`, `having`, and `orderBy`) should make a lot of sense for those familiar with SQL, but for those who aren't, here's a quick explanation of each:

- `groupBy` – adding this will allow you to group the results by a specified column(s). This would come in handy if you needed to obtain, say, a table with course IDs and the number of students enrolled in that course: simply grouping by course ID in the `Class` table would accomplish this.
- `having` – used in conjunction with a `groupBy` clause, this clause allows you to filter the aggregated results. Say you grouped by course ID in the `Class` table and wanted to filter out all classes with having less than 10 students enrolled, you could accomplish this with the `having` clause.
- `orderBy` – a fairly straightforward clause to use, the `orderBy` clause allows us to sort our query's resulting sub table by a specified column(s) and by ascending or descending order. For instance, say you wanted to sort the `Students` table by grade and then by name – specifying an `orderBy` clause would allow you to do this.

Lastly, in the two `query()` variants, you'll see the added parameters `limit` and `distinct`: the `limit` parameter allows you to limit how many rows you want back, and the `distinct` boolean allows you to specify whether you only want to return distinct rows. If this still doesn't make too much sense to you, no fears – we'll focus on building complex queries in the next chapter.

Now that we understand how the `query()` method works, let's revisit our earlier example and flush out the `getStudentsByGradeForCourse()` method. Though there are many ways to execute this method, conceptually they are all very similar: first, we query for all the students in the given course, and then of these students we want to filter and only keep those in the specified grade. The way I implemented it was by first obtaining a set of all student IDs from the given course, then obtaining a set of all the students for the given grade, and simply returning the intersection of those two sets. As for whether or not this is the optimal implementation simply depends on the size of your database.

And now, last but not least, let's enforce those removal rules mentioned earlier with some special remove wrapper methods:

```
public class SchemaHelper extends SQLiteOpenHelper {

    public Cursor getStudentsForCourse(int courseId) {
        ...
    }

    public Cursor getCoursesForStudent(int studentId) {
        ...
    }

    public Set<Integer> getStudentsAndGradeForCourse(int courseId,
int grade) {
        ...
    }

    // METHOD FOR SAFELY REMOVING A STUDENT
    public boolean removeStudent(int studentId) {
        SQLiteDatabase sd = getWritableDatabase();
        String[] whereArgs = new String[] { String.valueOf(studentId)
};

        // DELETE ALL CLASS MAPPINGS STUDENT IS SIGNED UP FOR
        sd.delete(ClassTable.TABLE_NAME, ClassTable.STUDENT_ID +
        "= ? ", whereArgs);

        // THEN DELETE STUDENT
        int result = sd.delete(StudentTable.TABLE_NAME,
        StudentTable.ID + "= ? ", whereArgs);
        return (result > 0);
    }

    // METHOD FOR SAFELY REMOVING A STUDENT
    public boolean removeCourse(int courseId) {
        SQLiteDatabase sd = getWritableDatabase();
        String[] whereArgs = new String[] { String.valueOf(courseId)
};

        // MAKE SURE YOU REMOVE COURSE FROM ALL STUDENTS ENROLLED
        sd.delete(ClassTable.TABLE_NAME, ClassTable.COURSE_ID +
        "= ? ", whereArgs);

        // THEN DELETE COURSE
        int result = sd.delete(CourseTable.TABLE_NAME,
        CourseTable.ID + "= ? ", whereArgs);
        return (result > 0);
    }
}
```

So here we have two remove methods, and in each one we manually enforce some schema rules by preventing someone from dropping a course without first removing those courses from the `Class` mapping table and vice versa. We call the `SQLiteDatabase` class's `delete()` method which, much like the `query()` method, allows you to pass in the table name, specify a filter argument (that is, a `WHERE` clause), and then allows you to pass in those filters' values (note that in both the `delete()` and `query()` methods, you can specify multiple filters, but more on this later).

Finally, let's put these methods in action and implement an `Activity` class:

```
public class SchemaActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        SchemaHelper sh = new SchemaHelper(this);

        // ADD STUDENTS AND RETURN THEIR IDS
        long sid1 = sh.addStudent("Jason Wei", "IL", 12);
        long sid2 = sh.addStudent("Du Chung", "AR", 12);
        long sid3 = sh.addStudent("George Tang", "CA", 11);
        long sid4 = sh.addStudent("Mark Bocanegra", "CA", 11);
        long sid5 = sh.addStudent("Bobby Wei", "IL", 12);

        // ADD COURSES AND RETURN THEIR IDS
        long cid1 = sh.addCourse("Math51");
        long cid2 = sh.addCourse("CS106A");
        long cid3 = sh.addCourse("Econ1A");

        // ENROLL STUDENTS IN CLASSES
        sh.enrollStudentClass((int) sid1, (int) cid1);
        sh.enrollStudentClass((int) sid1, (int) cid2);
        sh.enrollStudentClass((int) sid2, (int) cid2);
        sh.enrollStudentClass((int) sid3, (int) cid1);
        sh.enrollStudentClass((int) sid3, (int) cid2);
        sh.enrollStudentClass((int) sid4, (int) cid3);
        sh.enrollStudentClass((int) sid5, (int) cid2);

        // GET STUDENTS FOR COURSE
        Cursor c = sh.getStudentsForCourse((int) cid2);
        while (c.moveToNext()) {
            int colid = c.getColumnIndex(ClassTable.STUDENT_ID);
            int sid = c.getInt(colid);
```

```

        System.out.println("STUDENT " + sid + "
        IS ENROLLED IN COURSE " + cid2);
    }

    // GET STUDENTS FOR COURSE AND FILTER BY GRADE
    Set<Integer> sids = sh.getStudentsByGradeForCourse
    ((int) cid2, 11);
    for (Integer sid : sids) {
        System.out.println("STUDENT " + sid +
        " OF GRADE 11 IS ENROLLED IN COURSE " + cid2);
    }
}
}

```

So first we add some dummy data into our schema; in my case, I will add five students and three courses, and then enroll those students into some classes. Once the schema has some data in it, I will try out some methods and first request all the students signed up for CS106A. Afterwards, I will test another wrapper method we wrote and request all the students signed up for CS106A, but this time only those students in grade 11. And so the output from running this Activity is as follows:

tag	Message
System.err	Can't dispatch DDM chunk 46454154: no handler defined
System.err	
LOG_TAG	
System.out	STUDENT 1 IS ENROLLED IN COURSE 2
System.out	STUDENT 2 IS ENROLLED IN COURSE 2
System.out	STUDENT 3 IS ENROLLED IN COURSE 2
System.out	STUDENT 5 IS ENROLLED IN COURSE 2
System.out	STUDENT 3 OF GRADE 11 IS ENROLLED IN COURSE 2
InputManagerService	
IInputConnectionVr...	
ActivityManager	Displayed activity: imei.apps.dataforandroid/.ch2.SchemaActivity: 1243 ms

And voila! We quickly see that Students 1, 2, 3, and 5 were all enrolled in CS106A. However, after filtering by grade 11, we see that Student 3 is the only one signed up for CS106A in grade 11 – poor George. Now let's test out the remove methods:

```

public class SchemaActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        SchemaHelper sh = new SchemaHelper(this);

        long sid1 = sh.addStudent("Jason Wei", "IL", 12);
    }
}

```

```

// GET CLASSES I'M TAKING
c = sh.getCoursesForStudent((int) sid1);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(ClassTable.COURSE_ID);
    int cid = c.getInt(colid);
    System.out.println("STUDENT " + sid1 +
        " IS ENROLLED IN COURSE " + cid);
}

// TRY REMOVING A COURSE
sh.removeCourse((int) cid1);

System.out.println("-----");

// SEE IF REMOVAL KEPT SCHEMA CONSISTENT
c = sh.getCoursesForStudent((int) sid1);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(ClassTable.COURSE_ID);
    int cid = c.getInt(colid);
    System.out.println("STUDENT " + sid1 +
        " IS ENROLLED IN COURSE " + cid);
}
}
}

```

This time around, we first ask for all the classes that Student 1 (myself) is enrolled in. But oh no! Something happened to Math51 this quarter and so it was cancelled! We remove the course and make another request to look at all the classes that Student 1 is enrolled in – expecting to see that Math51 has been removed from the list. The output is as follows:

System.out	
System.out	
System.out	
System.out	STUDENT 1 IS ENROLLED IN COURSE 1
System.out	STUDENT 1 IS ENROLLED IN COURSE 2
System.out	-----
System.out	STUDENT 1 IS ENROLLED IN COURSE 2
InputManagerService	
IInputConnectionWr...	
ActivityManager	Displayed activity jwei.apps.dataforandroid/.ch2.SchemaActivity:

Indeed, we see that at first I was enrolled in both Math51 and CS106A, but after the course was removed, I'm not only enrolled in CS106A! By putting wrappers around some of these common insert, get, and remove functions, we can both simplify our development lives going forward while also enforcing certain schema rules.

Finally, let's conclude this chapter with how you can hook into a SQLite terminal to look at your data in table form and also issue SQLite queries – something extremely useful when debugging your application and making sure that your data is being added/updated/removed correctly.

Debugging your SQLite database

The Android platform provides you with a very powerful debugging friend called the **Android Debug Bridge (adb)**. The adb shell is a versatile command-line interface that allows you to communicate with a running emulator or a connected Android device. The adb tool can be found in your SDK's `/platform-tools` directory and, once, started is capable of doing everything from installing applications, to pushing and pulling data from your emulator, to plugging into your sqlite3 database and issuing queries (see the developer docs <http://developer.android.com/guide/developing/tools/adb.html> for more details).

In order to use adb, simply open your terminal and navigate to `/<your-sdk-directory>/platform-tools/` and type the following command:

```
adb shell
```

or type the following command if you want to target a specific emulator to connect to:

```
adb -s emulator-xxxx shell
```

At this point, you should have started the adb tool, at which point you need to tell it to connect to the emulator's sqlite3 database. This can be done by issuing the command `sqlite3` and then passing the path to your application's database file as follows:

```
# sqlite3 /data/data/<your-package-path>/databases/<your-database>.db
```

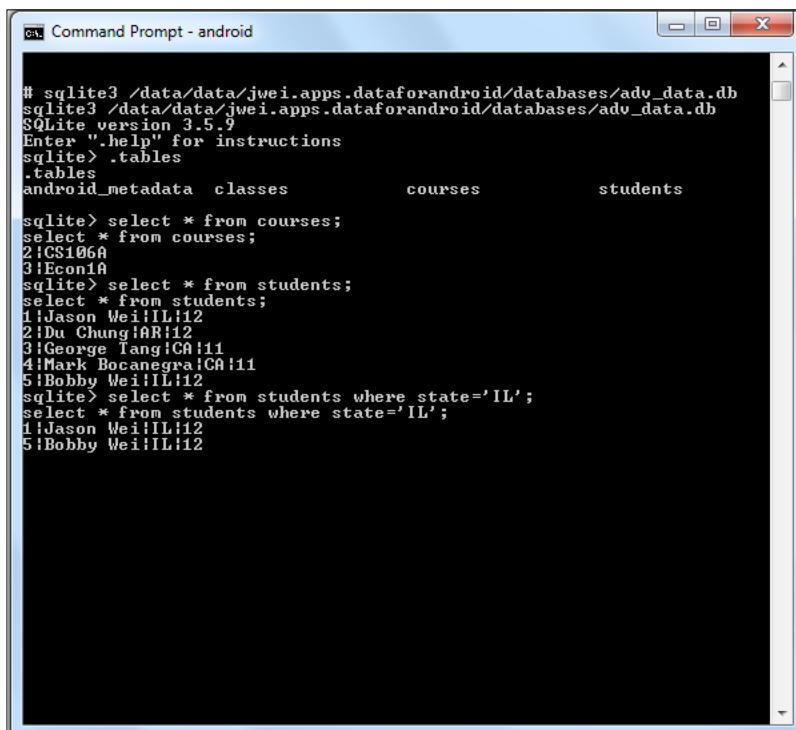
In our case, the command looked like the following:

```
# sqlite3 /data/data/jwei.apps.dataforandroid/databases/adv_data.db
```

And at this point, we should be able to issue all kinds of SQL queries that allow us to do everything from looking at our database schema to updating and removing individual rows of data in any of our tables. Some sample commands that you'll probably find most useful are as follows:

- `.tables` – shows you a list of all the tables in your database
- `.output FILENAME` – allows you to output the results of a query into a file (say, for further analysis)
- `.mode MODE` – allows you to specify the output file format (that is, as a CSV, HTML, and so on, could be useful for spreadsheet type analytics)
- `SELECT * FROM table_name` – standard query for selecting all columns of a given table (this is equivalent to a `get()` command for rows of a table)
- `SELECT * FROM table_name WHERE col = 'value'` – standard query for selecting all columns of a given table but with a column filter
- `SELECT col1, col2 FROM table_name` – standard query for selecting specific columns of a given table

And here's an example of us putting some of these commands to use with our previous schema:



```

C:\> sqlite3 /data/data/jwei.apps.dataforandroid/databases/adv_data.db
sqlite3 /data/data/jwei.apps.dataforandroid/databases/adv_data.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .tables
.tables
android_metadata  classes          courses          students

sqlite> select * from courses;
select * from courses;
2|CS106A
3|Econ1A
sqlite> select * from students;
select * from students;
1|Jason Wei|IL|12
2|Du Chung|AR|12
3|George Tang|CA|11
4|Mark Bocanegra|CA|11
5|Bobby Wei|IL|12
sqlite> select * from students where state='IL';
select * from students where state='IL';
1|Jason Wei|IL|12
5|Bobby Wei|IL|12

```

Hopefully this should get you going, but for a full list of sqlite3 commands just check out <http://www.sqlite.org/sqlite.html>, and for a more extensive list of complex queries just stay put – it's coming up next.

Summary

In this chapter, we went from a super bare-bones database schema that just contained one table to an entire schema containing multiple interdependent tables. We first saw how to create and upgrade multiple tables through overriding the `SQLiteOpenHelper` class, and then thought about some of the challenges surrounding a database schema with interdependencies. We decided to tackle these challenges by surrounding our database schema and its tables with an army of wrapper methods, designed for both ease of future development, as well as robustness in future data. These wrapper methods included everything from simple add methods, helpful as we were able to conceal the need to constantly request a writeable `SQLiteDatabase`, to more complex remove methods which concealed all of the functionality needed for enforcing various schema rules.

Then, we actually implemented an `Activity` class to show off our new database schema and ran through some sample database commands to test their functionality. Though we were able to validate and output the results of all our commands, we realized that this was a pretty verbose and suboptimal way for debugging our sqlite3 database, and so we looked into the Android Debug Bridge (adb) tool. With the adb tool, we were able to open a command-line terminal that then hooked into a running instance of an emulator or Android device, and subsequently, connect to that emulator/device's sqlite3 database. Here we were able to interact with the sqlite3 database in a very natural way by issuing various SQL commands and queries.

Now, the queries that we've seen so far have been pretty elementary, but if necessary, will do the trick for the majority of your application development needs. However, we'll see in the next chapter that by mastering more advanced SQL query concepts, we can enjoy both a substantial performance boost as well as a substantial memory boost in our application!

3

SQLite Queries

In the last chapter, we kicked our database building up a notch – transforming a simple schema involving just one, lone table, into a complex schema involving three interdependent tables. And now that we have a solid foundation in developing custom SQLite databases for Android, it's time to put the icing on the cake.

Though in theory, we could have one universal `get()` query which returns to us all columns of every row in our database as a `Cursor` object, and then filter and manipulate each row for our desired data – we can do better. Don't get me wrong – Java is fast – but when it comes to dealing with potentially thousands of rows of data on relatively limited memory, why not optimize things and let SQL do what it does best – that is, query for things!

In this next chapter, we will focus on striking the right balance between parsing and filtering your data on the Android client side (that is, with the Java interface), and building a more advanced SQL query and parsing/filtering your data in the SQLite database itself.

Methods for building SQLite queries

First, let's establish the different ways in which we can build a query. Just like we saw earlier, the most low-level method for querying the SQLite database is through the `SQLiteDatabase` class's `rawQuery()` method, defined as follows:

```
Cursor rawQuery(String sql, String[] selectionArgs)
```

This method is primarily for those with a strong background in SQL, as you pass SQL queries directly into the method as the first parameter. If your SQL query involves any sort of `WHERE` filter, then the second parameter allows you to pass in these filter values (we'll see several examples of this in use soon).

The second query method the `SQLiteDatabase` class provides you with is a convenience wrapper for submitting queries – with the `query()` method (something we also saw earlier) any actual SQL programming is hidden and, instead, all parts of the query are passed in as parameters:

```
Cursor query(String table, String[] columns, String selection,
String[] selectionArgs, String groupBy, String having, String orderBy)
```

With alternative `query()` methods containing parameters for `distinct` and `limit` constraints. Again, the previous parameters should be relatively self-explanatory, but all these methods will make the most sense when seen together for one given query. However, before moving on to those examples, let's take a look at the third method for building SQL queries.

This third method is one we haven't seen yet and comes from the `SQLiteQueryBuilder` class. Instead of having to submit raw SQL queries, or having to deal with convenience methods, which still may seem intimidating to those completely new to SQL, the Android platform decided to provide an entire convenience class to help developers interact with their SQLite databases as seamlessly as possible. Though this class has many methods associated with it (and I invite you to browse the developer docs online for more details), the following are some of the more important methods that we'll be highlighting later in this chapter:

```
String buildQuery(String[] projectionIn, String selection, String
groupBy, String having, String sortOrder, String limit)
```

The previous method is a convenience method for constructing a `SELECT` statement, which can be used then for a group of `SELECT` statements that will be joined through a `UNION` operator in the `buildUnionQuery()` method as follows:

```
String buildUnionQuery(String[] subQueries, String sortOrder,
String limit)
```

A method which allows you to pass in a set of `SELECT` statements (potentially constructed using the `buildQuery()` convenience method) and constructs a query that will return the `UNION` of those subqueries is as follows:

```
String buildQueryString(boolean distinct, String tables, String[]
columns, String where, String groupBy, String having, String orderBy,
String limit)
```

Builds a SQL query with the given parameters, similar to the `SQLiteDatabase` class's `query()` method but simply returns the query as a `String`:

```
Void setDistinct(boolean distinct)
```

The above allows you to set your current query as `DISTINCT` rows only.

```
Void setTables(String inTables)
```

Allows you to set the list of tables to query and if multiple tables are passed, in then it allows you to perform a `JOIN` on those tables.

So now that we have a list of all the different methods available to us, let's explore some basic SQLite queries and look at how we would perform relatively simple queries using each of the methods described previously!

SELECT statements

Using our `Students` schema from *Chapter 2, Using a SQLite Database*, let's begin with a glimpse at what our `Students` table looks like at this point:

Id	Name	State	Grade
1	Jason Wei	IL	12
2	Du Chung	AR	12
3	George Tang	CA	11
4	Mark Bocanegra	CA	11
5	Bobby Wei	IL	12

In this way, for each query that we do, we'll know exactly what results we should expect and, thus, we can validate our queries. Before we dive right, in here's a list of what we'll cover in this section:

- `SELECT` statements
- `SELECT` statements with column specifications
- `WHERE` filters
- `AND/OR` operators
- `DISTINCT` clause
- `LIMIT` clause

It'll be a lot to take in at once, especially for those with no prior SQL experience, but once you learn these basic building blocks, you'll be well on your way to building longer, more complex queries. And so, let's begin with the most basic `SELECT` query:

```
public class BasicQueryActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
super.onCreate(savedInstanceState);
setContentView(R.layout.main);

/*
 * SELECT Query
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
Cursor c = sqdb.rawQuery("SELECT * from " +
StudentTable.TABLE_NAME, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.NAME);
    String name = c.getString(colid);
    System.out.println("GOT STUDENT " + name);
}

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
c = sqdb.query(StudentTable.TABLE_NAME, null, null,
null, null, null, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.NAME);
    String name = c.getString(colid);
    System.out.println("GOT STUDENT " + name);
}

System.out.println("METHOD 3");

// METHOD #3 - SQLITEQUERYBUILDER
String query = SQLiteQueryBuilder.buildQueryString
(false, StudentTable.TABLE_NAME, null, null, null, null,
null, null);
System.out.println(query);
c = sqdb.rawQuery(query, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.NAME);
    String name = c.getString(colid);
    System.out.println("GOT STUDENT " + name);
}
}
}
```

Here, we see that in the first method, we're simply passing in the standard SQL query, while in the second method we are breaking down the query into its different parameters (that is, its table name, its selection filters, and so on). Finally, in the last method, which we notice looks very similar to the second method (for now), we again break down the query into its different parameters, but instead of returning a `Cursor`, our method returns the query as a `String`, which we can then execute as a raw query. The reasoning behind this is that one of `SQLiteQueryBuilder`'s strengths is that you can specify multiple queries and submit them all at the same time and effectively perform a `UNION` SQL query – but again we will play with this functionality later.

Now, let's take a look at the results from those queries and see if we can validate the results:

The screenshot shows the LogCat interface with tabs for LogCat, Outline, Properties, and Console. The LogCat tab is active, displaying a list of log messages. The messages are organized into columns: Time, pid, tag, and Message. The log messages show three methods being executed, each returning a list of student names. The third method also shows the raw SQL query: SELECT * FROM students. The log messages are as follows:

Time	pid	tag	Message
12-31 10:49:14.402	I	8047	System.out METHOD 1
12-31 10:49:14.412	I	8047	System.out GOT STUDENT Jason Vei
12-31 10:49:14.412	I	8047	System.out GOT STUDENT Du Chung
12-31 10:49:14.423	I	8047	System.out GOT STUDENT George Tang
12-31 10:49:14.423	I	8047	System.out GOT STUDENT Mark Bocanegra
12-31 10:49:14.423	I	8047	System.out GOT STUDENT Bobby Vei
12-31 10:49:14.423	I	8047	System.out METHOD 2
12-31 10:49:14.433	I	8047	System.out GOT STUDENT Jason Vei
12-31 10:49:14.433	I	8047	System.out GOT STUDENT Du Chung
12-31 10:49:14.433	I	8047	System.out GOT STUDENT George Tang
12-31 10:49:14.443	I	8047	System.out GOT STUDENT Mark Bocanegra
12-31 10:49:14.443	I	8047	System.out GOT STUDENT Bobby Vei
12-31 10:49:14.443	I	8047	System.out METHOD 3
12-31 10:49:14.443	I	8047	System.out SELECT * FROM students
12-31 10:49:14.443	I	8047	System.out GOT STUDENT Jason Vei
12-31 10:49:14.443	I	8047	System.out GOT STUDENT Du Chung
12-31 10:49:14.443	I	8047	System.out GOT STUDENT George Tang
12-31 10:49:14.453	I	8047	System.out GOT STUDENT Mark Bocanegra
12-31 10:49:14.453	I	8047	System.out GOT STUDENT Bobby Vei
12-31 10:49:14.623	I	19280	ActivityManager Displayed activity jwei.apps.dataforandroid/.ch3.BasicQueryActivity: 818 ms

Looks pretty good to me! We see that each method was able to return all rows of our table as expected. Under the third method, we can also see the query that was constructed using our `SQLiteQueryBuilder` class and indeed verify that the SQL query we submitted in the first method matches that built-in the third method.

Now, say you have a large table with thousands of rows of data and with tens of columns – for the sake of both efficiency and memory, it's often suggested in practice that you don't return the entire table with your queries but, instead, refine your queries to only return those columns of data of interest! And so, let's take a look at how we can specify which columns to return in our `SELECT` queries:

```
/*
 * SELECT COLUMNS Query
 */

System.out.println("METHOD 1");
```



```
// METHOD #1 - SQLITEDATABASE RAWQUERY()
c = sqdb.rawQuery(
    "SELECT " + StudentTable.NAME + "," + StudentTable.STATE + " from "
    + StudentTable.TABLE_NAME, null);
    while (c.moveToNext()) {
        int colid = c.getColumnIndex(StudentTable.NAME);
        int colid2 = c.getColumnIndex(StudentTable.STATE);

    }

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
String[] cols = new String[] { StudentTable.NAME, StudentTable.STATE
};
c = sqdb.query(StudentTable.TABLE_NAME, cols, null, null, null,
null, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.NAME);
    int colid2 = c.getColumnIndex(StudentTable.STATE);

}

System.out.println("METHOD 3");

// METHOD #3 - SQLITEQUERYBUILDER
query = SQLiteQueryBuilder.buildQueryString(false, StudentTable.TABLE_
NAME, cols, null, null, null, null, null);
System.out.println(query);
c = sqdb.rawQuery(query, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.NAME);
    int colid2 = c.getColumnIndex(StudentTable.STATE);

}

}
```

And so, we see that the overall structure of the query is the same for all three methods, but in methods two and three, we pass in a `String[]` containing the columns of data that we want. Again, just to verify that our queries are behaving the way we want them to, here's the output of those queries:

Log				
Time		pid	tag	Message
12-31 10:54:59.932	I	8216	System.out	METHOD 1
12-31 10:54:59.942	I	8216	System.out	GOT STUDENT Jason Wei FROM IL
12-31 10:54:59.942	I	8216	System.out	GOT STUDENT Du Chung FROM AR
12-31 10:54:59.942	I	8216	System.out	GOT STUDENT George Tang FROM CA
12-31 10:54:59.942	I	8216	System.out	GOT STUDENT Mark Bocanegra FROM CA
12-31 10:54:59.942	I	8216	System.out	GOT STUDENT Bobby Wei FROM IL
12-31 10:54:59.942	I	8216	System.out	METHOD 2
12-31 10:54:59.952	I	8216	System.out	GOT STUDENT Jason Wei FROM IL
12-31 10:54:59.952	I	8216	System.out	GOT STUDENT Du Chung FROM AR
12-31 10:54:59.952	I	8216	System.out	GOT STUDENT George Tang FROM CA
12-31 10:54:59.952	I	8216	System.out	GOT STUDENT Mark Bocanegra FROM CA
12-31 10:54:59.952	I	8216	System.out	GOT STUDENT Bobby Wei FROM IL
12-31 10:54:59.952	I	8216	System.out	METHOD 3
12-31 10:54:59.952	I	8216	System.out	SELECT student_name, state FROM students
12-31 10:54:59.962	I	8216	System.out	GOT STUDENT Jason Wei FROM IL
12-31 10:54:59.962	I	8216	System.out	GOT STUDENT Du Chung FROM AR
12-31 10:54:59.962	I	8216	System.out	GOT STUDENT George Tang FROM CA
12-31 10:54:59.962	I	8216	System.out	GOT STUDENT Mark Bocanegra FROM CA
12-31 10:54:59.972	I	8216	System.out	GOT STUDENT Bobby Wei FROM IL

And so we see that indeed we are able to return each student, along with their respective states. Finally again, notice the query that is constructed in the third method and compare it to the raw SQL query that was passed to the first method – they should match exactly and they do.

WHERE filters and SQL operators

Now, oftentimes it's important to be able to filter your data not just by columns but also by column values! This is where the `WHERE` filter comes in handy and these `WHERE` filters will definitely be the most-used clause you will run into as a database developer. On that note, let's take a look at how these `WHERE` filters (also known as selection parameters in Android) are implemented with our three query-building methods:

```
/*
 * WHERE Filter - Filter by State
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
c = sqdb.rawQuery("SELECT * from " + StudentTable.TABLE_NAME + " WHERE
" + StudentTable.STATE + "= ? ", new String[] { "IL" });
while (c.moveToNext()) {
```

```
        int colid = c.getColumnIndex(StudentTable.NAME);
        int colid2 = c.getColumnIndex(StudentTable.STATE);

    }

    System.out.println("METHOD 2");
    // METHOD #2 - SQLITEDATABASE QUERY()
    c = sqdb.query(StudentTable.TABLE_NAME, null, StudentTable.STATE + "=
? ", new String[] { "IL" }, null, null, null);
    while (c.moveToNext()) {
        int colid = c.getColumnIndex(StudentTable.NAME);
        int colid2 = c.getColumnIndex(StudentTable.STATE);

    }

    System.out.println("METHOD 3");

    // METHOD #3 - SQLITEQUERYBUILDER
    query = SQLiteQueryBuilder.buildQueryString(false, StudentTable.TABLE_
NAME, null, StudentTable.STATE + "'IL'", null, null, null, null);
    System.out.println(query);
    c = sqdb.rawQuery(query, null);
    while (c.moveToNext()) {
        int colid = c.getColumnIndex(StudentTable.NAME);
        int colid2 = c.getColumnIndex(StudentTable.STATE);

    }
}
```

With the first method, we can see how a standard SQL `WHERE` clause is formatted. Knowing this, with our second and third methods we see that, we can just pass into the selection parameter a string formatted like the `WHERE` clause but omitting the `WHERE` itself (this is automatically appended to your query for you). This can explicitly be seen with the constructed query returned by our `SQLiteQueryBuilder` class in the third method:

Log				
Time	pid	tag	Message	
12-31 11:06:35.722	I	8524	System.out	METHOD 1
12-31 11:06:35.754	I	8524	System.out	GOT STUDENT Jason Wei FROM IL
12-31 11:06:35.762	I	8524	System.out	GOT STUDENT Bobby Wei FROM IL
12-31 11:06:35.762	I	8524	System.out	METHOD 2
12-31 11:06:35.772	I	8524	System.out	GOT STUDENT Jason Wei FROM IL
12-31 11:06:35.772	I	8524	System.out	GOT STUDENT Bobby Wei FROM IL
12-31 11:06:35.772	I	8524	System.out	METHOD 3
12-31 11:06:35.772	I	8524	System.out	SELECT * FROM students WHERE state='IL'
12-31 11:06:35.782	I	8524	System.out	GOT STUDENT Jason Wei FROM IL
12-31 11:06:35.782	I	8524	System.out	GOT STUDENT Bobby Wei FROM IL
12-31 11:06:35.962	I	18200	ActivityManager	Displayed activity jwei.apps.dataforandroid/.ch3.BasicQueryActivity: 920 ms

Just like with any programming language, you can filter logic through the use of AND/OR operators; the same applies to SQL and, specifically, with SQL WHERE filters. Instead of asking for all rows which satisfy one set of conditions, you can write queries which would return rows that satisfy all given conditions, or more loosely, just one of several given conditions. An example of this is as follows, where instead of only returning students from Illinois, we utilize the SQL OR operator and also ask for students from Arkansas:

```

/*
 * AND/OR Clauses
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
c = sqdb.rawQuery("SELECT * from " + StudentTable.TABLE_NAME + " WHERE
" + StudentTable.STATE + " = ? OR " + StudentTable.STATE + " = ?", new
String[] { "IL", "AR" });

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
c = sqdb.query(StudentTable.TABLE_NAME, null, StudentTable.STATE +
" = ? OR " + StudentTable.STATE + " = ?", new String[] { "IL", "AR" },
null, null, null);

System.out.println("METHOD 3");

// METHOD #3 - SQLITEQUERYBUILDER
query = SQLiteQueryBuilder.buildQueryString(false, StudentTable.TABLE_
NAME, null, StudentTable.STATE + " ='IL' OR " + StudentTable.STATE +
" ='AR'", null, null, null, null);
System.out.println(query);
c = sqdb.rawQuery(query, null);

```

Here you'll notice that the syntax is, again, very similar to the earlier example, but this time we've injected an OR operator into the WHERE filter (selection parameter) and have placed two selection arguments (that is, the '?') instead of one. It's important to note that the order of arguments contained in your `String[]` is important – more specifically, that the first `String` in your array will correspond to the first '?' place card, and so on. And of course, if you want to use the AND operator, then just apply the previous syntax but replacing OR with AND. Taking a quick peak at the output, we see as follows:

Time	pid	tag	Message
12-31 22:19:51.962	I	24771	System.out
12-31 22:19:51.972	I	24771	System.out
12-31 22:19:51.972	I	24771	System.out
12-31 22:19:51.972	I	24771	System.out
12-31 22:19:51.972	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:51.982	I	24771	System.out
12-31 22:19:52.443	I	19200	ActivityManager

And so now our buddy Du has popped into the result set!

DISTINCT and LIMIT clauses

Powering on, let's take a look at the DISTINCT clause in SQL:

```

/*
 * DISTINCT Clause
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
c = sqdb.rawQuery("SELECT DISTINCT " + StudentTable.STATE + " from " +
StudentTable.TABLE_NAME, null);

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
// SWITCH TO MORE GENERAL QUERY() METHOD
c = sqdb.query(true, StudentTable.TABLE_NAME, new String[] {
StudentTable.STATE }, null, null, null, null, null, null);

...

```

```

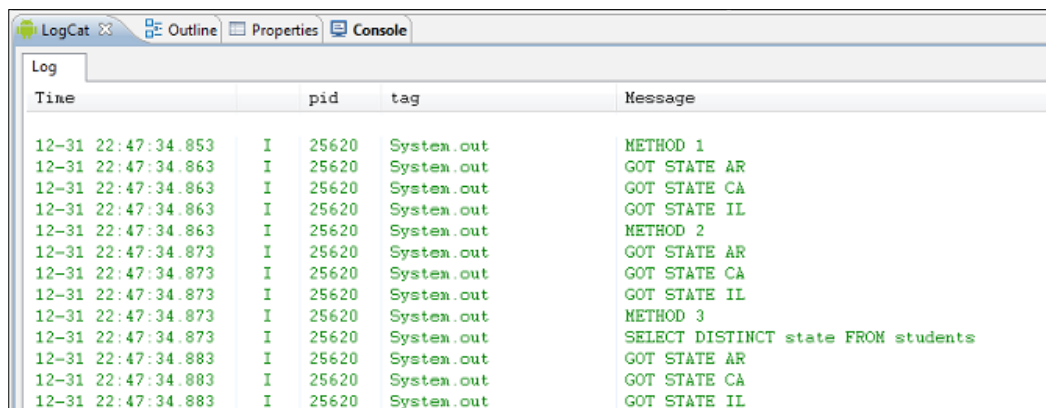
System.out.println("METHOD 3");

// METHOD #3 - SQLITEQUERYBUILDER
query = SQLiteQueryBuilder.buildQueryString(true, StudentTable.TABLE_
NAME, new String[] { StudentTable.STATE }, null, null, null, null,
null);
System.out.println(query);
c = sqdb.rawQuery(query, null);

```

The `DISTINCT` clause is also relatively straightforward – it allows you to specify in your query that for the given columns you only want to return a subset of rows which have distinct values for that column. Notice that I emphasize for the given columns, as in order for the `DISTINCT` clause to be meaningful, a column must be specified in your query.

In my previous example, we'll notice a couple of things. First off, in our query, notice that we follow the `DISTINCT` clause with the column that we want it to apply to – namely the `state` column. Essentially, we're asking my query to return to us a subtable with all of the distinct states in my database. Said another way, we want to know what states our students come from and only want one row per state. Another thing worth mentioning is that we've switched the `query()` statement that we were previously using in the second method – this time switching it to a more general `query()` method which has parameters for specifying a `DISTINCT` clause. The results for this query were:



Time	pid	tag	Message
12-31 22:47:34.853	I	25620	System.out METHOD 1
12-31 22:47:34.863	I	25620	System.out GOT STATE AR
12-31 22:47:34.863	I	25620	System.out GOT STATE CA
12-31 22:47:34.863	I	25620	System.out GOT STATE IL
12-31 22:47:34.863	I	25620	System.out METHOD 2
12-31 22:47:34.873	I	25620	System.out GOT STATE AR
12-31 22:47:34.873	I	25620	System.out GOT STATE CA
12-31 22:47:34.873	I	25620	System.out GOT STATE IL
12-31 22:47:34.873	I	25620	System.out METHOD 3
12-31 22:47:34.873	I	25620	System.out SELECT DISTINCT state FROM students
12-31 22:47:34.883	I	25620	System.out GOT STATE AR
12-31 22:47:34.883	I	25620	System.out GOT STATE CA
12-31 22:47:34.883	I	25620	System.out GOT STATE IL

Which is indeed the case for our current table! And last but not least, let's take a look at the `LIMIT` clause:

```
/*
 * LIMIT Clause
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
c = sqdb.rawQuery("SELECT * from " + StudentTable.TABLE_NAME + " LIMIT
0,3", null);

...

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
// SWITCH TO MORE GENERAL QUERY() METHOD
c = sqdb.query(false, StudentTable.TABLE_NAME, null, null, null, null,
null, null, "3");

System.out.println("METHOD 3");

// METHOD #3 - SQLITEQUERYBUILDER
query = SQLiteQueryBuilder.buildQueryString(false, StudentTable.TABLE_
NAME, null, null, null, null, null, "3");
System.out.println(query);
c = sqdb.rawQuery(query, null);
```

The `LIMIT` clause simply allows you to limit how many rows to return. The `LIMIT` clause takes on two formats:

- `LIMIT n, m`
- `LIMIT n`

The first format tells the query to return just `m` rows (that is, limiting how many rows to return) starting from row `n`. The second format simply tells the query to return the first `n` rows which satisfy the given conditions. The first format definitely provides us with more flexibility, but, unfortunately, neither the second nor the third method allows us to take advantage of this format (due to the way that it automatically constructs the query for us), while the first format (the raw SQL query) can execute any valid SQL query. This is a small example of the versatility that executing raw SQL queries gives us, and is a perfect example of trading versatility for convenience and abstraction. In any case, let's just make one last sanity check here to make sure our queries are actually only returning three rows:

Time	pid	tag	Message
12-31 22:47:34.803	I	25620	System.out
12-31 22:47:34.893	I	25620	System.out
12-31 22:47:34.893	I	25620	System.out
12-31 22:47:34.893	I	25620	System.out
12-31 22:47:34.903	I	25620	System.out
12-31 22:47:34.903	I	25620	System.out
12-31 22:47:34.913	I	25620	System.out
12-31 22:47:34.913	I	25620	System.out
12-31 22:47:34.913	I	25620	System.out
12-31 22:47:34.913	I	25620	System.out
12-31 22:47:34.913	I	25620	System.out
12-31 22:47:34.923	I	25620	System.out
12-31 22:47:34.923	I	25620	System.out
12-31 22:47:34.923	I	25620	System.out
12-31 22:47:35.113	I	10200	ActivityManager

Yup – looks good to me! In all methods, even though we didn't specify any `WHERE` filters, we were still only returned the first three valid results, as expected.

In this section, we looked at a number of clauses built-into the SQL language which allow us to have control over our data. By introducing these clauses one by one, the hope was that you could first see all of the pieces of the puzzle. Then, when the time comes for you to implement your own database, you'll be able to put the pieces together and execute powerful queries which quickly return meaningful data. However, before we wrap up this chapter, let's look at some advanced queries, which will take more time to master and understand, but again will add another tool under your belt.

ORDER BY and GROUP BY clauses

In this section, we'll look at some of the more advanced and more nuanced features of the SQL language as well as their implementations in the various SQL convenience classes of Android. Again, before we dive in and attack these features, here's a list of what we'll be covering in this next section:

- `ORDER BY` clauses
- `GROUP BY` clauses
- `HAVING` filters
- SQL Functions
- `JOINS`

So let's look at `ORDER BY` clauses in SQL:

```
public class AdvancedQueryActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```



```
        setContentView(R.layout.main);

        SchemaHelper sch = new SchemaHelper(this);
        SQLiteDatabase sqdb = sch.getWritableDatabase();

        /*
         * ORDER BY Clause
         */

        System.out.println("METHOD 1");

        // METHOD #1 - SQLITEDATABASE RAWQUERY()
        Cursor c = sqdb.rawQuery("SELECT * from " +
        StudentTable.TABLE_NAME + " ORDER BY " + StudentTable.STATE +
        " ASC", null);
        while (c.moveToNext()) {
            int colid = c.getColumnIndex(StudentTable.NAME);
            int colid2 = c.getColumnIndex(StudentTable.STATE);
            String name = c.getString(colid);
            String state = c.getString(colid2);
            System.out.println("GOT STUDENT " + name +
            " FROM " + state);
        }

        System.out.println("METHOD 2");

        // METHOD #2 - SQLITEDATABASE QUERY()
        c = sqdb.query(StudentTable.TABLE_NAME, null, null,
        null, null, null, StudentTable.STATE + " ASC");
        while (c.moveToNext()) {
            int colid = c.getColumnIndex(StudentTable.NAME);
            int colid2 = c.getColumnIndex(StudentTable.STATE);
            ...
        }

        System.out.println("METHOD 3");

        // METHOD #3 - SQLITEQUERYBUILDER
        String query = SQLiteQueryBuilder.buildQueryString
        (false, StudentTable.TABLE_NAME, null, null, null,
        null, StudentTable.STATE + " ASC", null);

        System.out.println(query);
        c = sqdb.rawQuery(query, null);
        while (c.moveToNext()) {
            int colid = c.getColumnIndex(StudentTable.NAME);
            int colid2 = c.getColumnIndex(StudentTable.STATE);
            ...
        }
    }
}
```

Here the syntax for the ORDERBY clause is:

```
ORDER BY your_column ASC|DESC
```

So in the first method, we see this syntax in action, and then in the latter two methods, we see that we simply need to pass in the column name followed by either ASC or DESC (as a String) into the ORDERBY parameter of the respective query methods. In the latter two methods, the syntax is essentially the same, and so I won't go into too much detail here, but the important part is simply to know the components of an SQL ORDERBY clause. In all three methods shown, we are sorting our resulting subtable by the state column, and so to validate our query, we check the output and see the following:

LogCat

Outline

Properties

Console

Log

Time		pid	tag	Message
01-03 15:32:07.350	I	1066	System.out	METHOD 1
01-03 15:32:07.360	I	1066	System.out	GOT STUDENT Du Chung FROM AR
01-03 15:32:07.370	I	1066	System.out	GOT STUDENT George Tang FROM CA
01-03 15:32:07.370	I	1066	System.out	GOT STUDENT Mark Bocanegra FROM CA
01-03 15:32:07.370	I	1066	System.out	GOT STUDENT Jason Wei FROM IL
01-03 15:32:07.370	I	1066	System.out	GOT STUDENT Bobby Wei FROM IL
01-03 15:32:07.380	I	1066	System.out	METHOD 2
01-03 15:32:07.390	I	1066	System.out	GOT STUDENT Du Chung FROM AR
01-03 15:32:07.390	I	1066	System.out	GOT STUDENT George Tang FROM CA
01-03 15:32:07.390	I	1066	System.out	GOT STUDENT Mark Bocanegra FROM CA
01-03 15:32:07.390	I	1066	System.out	GOT STUDENT Jason Wei FROM IL
01-03 15:32:07.390	I	1066	System.out	GOT STUDENT Bobby Wei FROM IL
01-03 15:32:07.390	I	1066	System.out	METHOD 3
01-03 15:32:07.390	I	1066	System.out	SELECT * FROM students ORDER BY state ASC
01-03 15:32:07.400	I	1066	System.out	GOT STUDENT Du Chung FROM AR
01-03 15:32:07.400	I	1066	System.out	GOT STUDENT George Tang FROM CA
01-03 15:32:07.400	I	1066	System.out	GOT STUDENT Mark Bocanegra FROM CA
01-03 15:32:07.410	I	1066	System.out	GOT STUDENT Jason Wei FROM IL
01-03 15:32:07.410	I	1066	System.out	GOT STUDENT Bobby Wei FROM IL

So, indeed, we see that the resulting rows are sorted in ascending order by the state. Furthermore, just like with the basic queries, we can see the outputted SQL query that is created by the SQLiteQueryBuilder class, and can verify that this is the same query that is executed in our first method.

Now, moving on to GROUPBY clauses:

```
/*
 * GROUP BY Clause
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
String colName = "COUNT(" + StudentTable.STATE + ")";
```

```
c = sqdb.rawQuery("SELECT " + StudentTable.STATE + "," +
+ colName + " from " + StudentTable.TABLE_NAME + " GROUP BY "
+ StudentTable.STATE, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.STATE);
    int colid2 = c.getColumnIndex(colName);
    String state = c.getString(colid);
    int count = c.getInt(colid2);
    System.out.println("STATE " + state + " HAS COUNT " +
count);
}

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
c = sqdb.query(StudentTable.TABLE_NAME, new String[] {
StudentTable.STATE, colName }, null, null,
StudentTable.STATE, null, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.STATE);
    int colid2 = c.getColumnIndex(colName);
}

System.out.println("METHOD 3");

// METHOD #3 - SQLITEQUERYBUILDER
query = SQLiteQueryBuilder.buildQueryString(false,
StudentTable.TABLE_NAME, new String[] { StudentTable.STATE,
colName }, null, StudentTable.STATE, null, null, null);

System.out.println(query);
c = sqdb.rawQuery(query, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.STATE);
    int colid2 = c.getColumnIndex(colName);
}
}
```

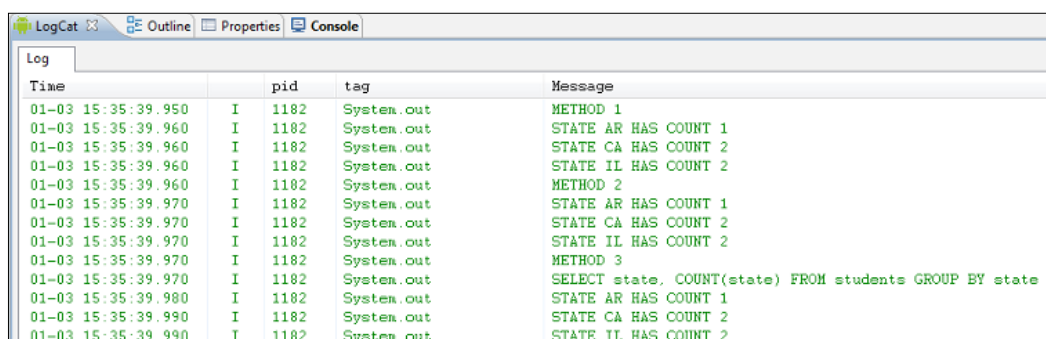
Now, it is again crucial to understand the structure of a GROUPBY query, as it is unlike any of the previous clauses or filters that we have seen. The structure is as follows:

```
SELECT your_column, aggregate_function(your_column) FROM your_table
GROUP BY your_column
```

The trickiest part is in the `aggregate_function(your_column)` segment of the query. In our case, we use what's known as the `COUNT()` function in SQL, which, as its name suggests, simply counts the number of rows returned in a query (or subquery) and returns the counted value. You can use any number of `aggregate_functions` in SQL, but for now let's stick with `COUNT()` and later when we discuss SQL Functions, I'll list out some of the others.

The idea here is simple – first we're selecting a column to group our data by (in our case, by state), and then we're telling the query to return two columns: the first is simply the states themselves, and the second is the number of times that state appears in our table (that is, the aggregate number of states in our table). You'll also notice that in both the second and third methods, the way the `GROUPBY` query is done is pretty simple, but the only tricky part is specifying the column name with the `COUNT()` function wrapped around it (see how we declare the String `colName`). Once you do that, the rest is straightforward and behaves just like a standard `SELECT` query with columns! Note that the `COUNT()` function also takes a `*` as a parameter, which simply returns a count of all the rows in the subtable.

And now, let's see what our output is:



Time	pid	tag	Message
01-03 15:35:39.950	I	System.out	METHOD 1
01-03 15:35:39.960	I	System.out	STATE AR HAS COUNT 1
01-03 15:35:39.960	I	System.out	STATE CA HAS COUNT 2
01-03 15:35:39.960	I	System.out	STATE IL HAS COUNT 2
01-03 15:35:39.960	I	System.out	METHOD 2
01-03 15:35:39.970	I	System.out	STATE AR HAS COUNT 1
01-03 15:35:39.970	I	System.out	STATE CA HAS COUNT 2
01-03 15:35:39.970	I	System.out	STATE IL HAS COUNT 2
01-03 15:35:39.970	I	System.out	METHOD 3
01-03 15:35:39.970	I	System.out	SELECT state, COUNT(state) FROM students GROUP BY state
01-03 15:35:39.980	I	System.out	STATE AR HAS COUNT 1
01-03 15:35:39.990	I	System.out	STATE CA HAS COUNT 2
01-03 15:35:39.990	I	System.out	STATE IL HAS COUNT 2

And voila! Just as we expected – our queries return each state followed by their respective frequencies!

HAVING filters and Aggregate functions

Now, with `GROUPBY` clauses come the `HAVING` filters. The `HAVING` filter is to be used only with a `GROUPBY` clause, and taking the previous queries as an example, say we want to group by the number of states in our table, but we only care about states that appear a certain number of times. With the `HAVING` filter, we can essentially phrase our query such that it groups by the number of states, and then only returns those states having a total count greater or less than some value.

Let's take a look at the following code and pay close attention to how I structure my query (it will look very similar to the `GROUP BY` query but with an extra filter at the end):

```
/*
 * HAVING Filter
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
String colName = "COUNT(" + StudentTable.STATE + ")";

c = sqdb.rawQuery("SELECT " + StudentTable.STATE + "," +
    colName + " from " + StudentTable.TABLE_NAME + " GROUP BY " +
    StudentTable.STATE + " HAVING " + colName + " > 1", null);

while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.STATE);
    int colid2 = c.getColumnIndex(colName);
}

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
c = sqdb.query(StudentTable.TABLE_NAME, new String[] {
    StudentTable.STATE, colName }, null, null, StudentTable.STATE,
    colName + " > 1", null);

System.out.println("METHOD 3");

// METHOD #3 - SQLITEQUERYBUILDER
query = SQLiteQueryBuilder.buildQueryString(false,
    StudentTable.TABLE_NAME, new String[] { StudentTable.STATE,
    colName }, null, StudentTable.STATE, colName + " > 1", null,
    null);
System.out.println(query);
c = sqdb.rawQuery(query, null);
```

And so you have it. Again, notice the structure of my query in the first method and notice how it translates into the `HAVING` parameter of the query convenience methods in the second and third methods. Let's see now how the query did and whether or not it eliminated AR from the output:

LogCat				
Log				
Time	pid	tag	Message	
01-03 15:39:18.870	I	1300	System.out	METHOD 1
01-03 15:39:18.890	I	1300	System.out	STATE CA HAS COUNT 2
01-03 15:39:18.890	I	1300	System.out	STATE IL HAS COUNT 2
01-03 15:39:18.890	I	1300	System.out	METHOD 2
01-03 15:39:18.900	I	1300	System.out	STATE CA HAS COUNT 2
01-03 15:39:18.900	I	1300	System.out	STATE IL HAS COUNT 2
01-03 15:39:18.900	I	1300	System.out	METHOD 3
01-03 15:39:18.900	I	1300	System.out	SELECT state, COUNT(state) FROM students GROUP BY state HAVING COUNT(state) > 1
01-03 15:39:18.910	I	1300	System.out	STATE CA HAS COUNT 2
01-03 15:39:18.910	I	1300	System.out	STATE IL HAS COUNT 2

Perfect – pretty straightforward. Earlier we ran into the `COUNT()` aggregate function, which along with `SUM()` and `AVG()` are amongst the most popular of the aggregate functions (see here for the full list: http://www.sqlite.org/lang_aggfunc.html). These functions, like their names suggest, either count the total number of rows returned in a particular column of your subtable, or sum of the values of that column in your subtable, or average of the values of that column in your subtable, and so on. First, let's examine some of these aggregate functions, listed as follows (notice how the column names change):

```
/*
 * SQL Functions - MIN/MAX/AVG
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
String colName = "MIN(" + StudentTable.GRADE + ")";

c = sqdb.rawQuery("SELECT " + colName + " from " +
StudentTable.TABLE_NAME, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(colName);
    int minGrade = c.getInt(colid);
    System.out.println("MIN GRADE " + minGrade);
}

System.out.println("METHOD 2");

// METHOD #2 - SQLITEDATABASE QUERY()
colName = "MAX(" + StudentTable.GRADE + ")";

c = sqdb.query(StudentTable.TABLE_NAME, new String[]
{ colName }, null, null, null, null, null);

System.out.println("METHOD 3");

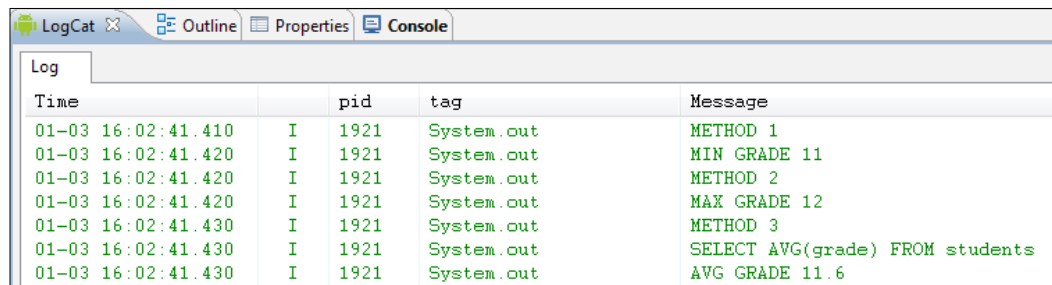
// METHOD #3 - SQLITEQUERYBUILDER
colName = "AVG(" + StudentTable.GRADE + ")";
```

```

query = SQLiteQueryBuilder.buildQueryString(false,
StudentTable.TABLE_NAME, new String[] { colName }, null,
null, null, null, null);
System.out.println(query);
c = sqdb.rawQuery(query, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(colName);
    double avgGrade = c.getDouble(colid);
    System.out.println("AVG GRADE " + avgGrade);
}

```

So, here we use each of the three methods to test out a different aggregate function. The results are shown as follows:



Time	pid	tag	Message
01-03 16:02:41.410	I 1921	System.out	METHOD 1
01-03 16:02:41.420	I 1921	System.out	MIN GRADE 11
01-03 16:02:41.420	I 1921	System.out	METHOD 2
01-03 16:02:41.420	I 1921	System.out	MAX GRADE 12
01-03 16:02:41.430	I 1921	System.out	METHOD 3
01-03 16:02:41.430	I 1921	System.out	SELECT AVG(grade) FROM students
01-03 16:02:41.430	I 1921	System.out	AVG GRADE 11.6

After referencing the state of the table from earlier, you can quickly validate the outputted numbers and confirm that the functions are indeed doing as they should. Outside of aggregate functions (which are typically used for numerical-typed columns), SQLite also provides you with an assortment of other core functions that help you manipulate everything from Strings to Date types, and so on. A complete list of these core functions can be found http://www.sqlite.org/lang_corefunc.html but for now, let's just take a look at a couple:

```

/*
 * SQL Functions - UPPER/LOWER/SUBSTR
 */

System.out.println("METHOD 1");

// METHOD #1 - SQLITEDATABASE RAWQUERY()
String colName = "UPPER(" + StudentTable.NAME + ")";

c = sqdb.rawQuery("SELECT " + colName + " from " +
StudentTable.TABLE_NAME, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(colName);

```

```

        String upperName = c.getString(colid);
        System.out.println("GOT STUDENT " + upperName);
    }

    System.out.println("METHOD 2");

    // METHOD #2 - SQLITEDATABASE QUERY()
    colName = "LOWER(" + StudentTable.NAME + ")";

    c = sqdb.query(StudentTable.TABLE_NAME, new String[]
    { colName }, null, null, null, null, null);

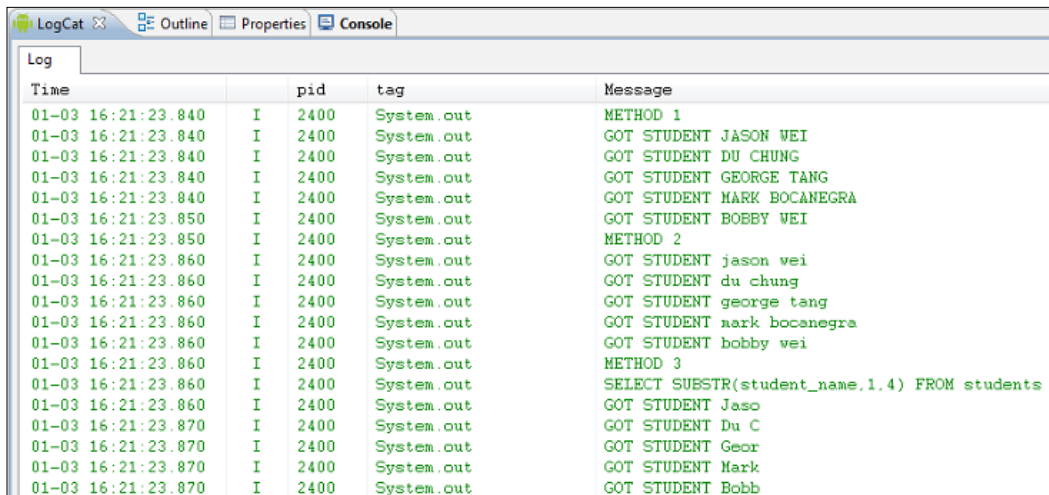
    System.out.println("METHOD 3");

    // METHOD #3 - SQLITEQUERYBUILDER
    colName = "SUBSTR(" + StudentTable.NAME + ",1,4)";

    query = SQLiteQueryBuilder.buildQueryString(false,
    StudentTable.TABLE_NAME, new String[] { colName }, null,
    null, null, null, null);
    System.out.println(query);
    c = sqdb.rawQuery(query, null);

```

Again, here is the associated output of these core functions:



Time	pid	tag	Message
01-03 16:21:23.840	I 2400	System.out	METHOD 1
01-03 16:21:23.840	I 2400	System.out	GOT STUDENT JASON WEI
01-03 16:21:23.840	I 2400	System.out	GOT STUDENT DU CHUNG
01-03 16:21:23.840	I 2400	System.out	GOT STUDENT GEORGE TANG
01-03 16:21:23.840	I 2400	System.out	GOT STUDENT MARK BOCANEGRA
01-03 16:21:23.850	I 2400	System.out	GOT STUDENT BOBBY WEI
01-03 16:21:23.850	I 2400	System.out	METHOD 2
01-03 16:21:23.860	I 2400	System.out	GOT STUDENT jason wei
01-03 16:21:23.860	I 2400	System.out	GOT STUDENT du chung
01-03 16:21:23.860	I 2400	System.out	GOT STUDENT george tang
01-03 16:21:23.860	I 2400	System.out	GOT STUDENT mark bocanegra
01-03 16:21:23.860	I 2400	System.out	GOT STUDENT bobby wei
01-03 16:21:23.860	I 2400	System.out	METHOD 3
01-03 16:21:23.860	I 2400	System.out	SELECT SUBSTR(student_name,1,4) FROM students
01-03 16:21:23.860	I 2400	System.out	GOT STUDENT Jaso
01-03 16:21:23.870	I 2400	System.out	GOT STUDENT Du C
01-03 16:21:23.870	I 2400	System.out	GOT STUDENT Geor
01-03 16:21:23.870	I 2400	System.out	GOT STUDENT Mark
01-03 16:21:23.870	I 2400	System.out	GOT STUDENT Bobb

Now, as far as how much of a performance boost running some of these functions in SQLite as opposed to just doing them on the Java side, this is debatable and is highly dependent on the size of your database and the function you are calling. For instance, some string manipulation functions may not offer as much of a performance boost as other more complex aggregate functions. In fact, this SQLite to Java comparison is something we'll look more into in the next section, but regardless, it's always better to be aware of the functions available to you in SQLite and add them to your arsenal of weapons!

And lastly, it's about time we looked at the benefits of using the `SQLiteQueryBuilder` (until now, much of the syntax was very similar to just the `query()` method in `SQLiteDatabase`) and see how we can leverage this convenience class to perform more complicated joins:

```
/*
 * SQL JOINS
 */

SQLiteQueryBuilder sqb = new SQLiteQueryBuilder();

// NOTICE THE SYNTAX FOR COLUMNS IN JOIN QUERIES
String courseIdCol = CourseTable.TABLE_NAME + "." +
    CourseTable.ID;

String classCourseIdCol = ClassTable.TABLE_NAME + "." +
    ClassTable.COURSE_ID;

String classIdCol = ClassTable.TABLE_NAME + "." +
    ClassTable.ID;

sqb.setTables(ClassTable.TABLE_NAME + " INNER JOIN " +
    CourseTable.TABLE_NAME + " ON (" + classCourseIdCol + " = "
    + courseIdCol + ")");

String[] cols = new String[]
{ classIdCol, ClassTable.COURSE_ID, CourseTable.NAME };

query = sqb.buildQuery(cols, null, null, null, null,
    null, null);

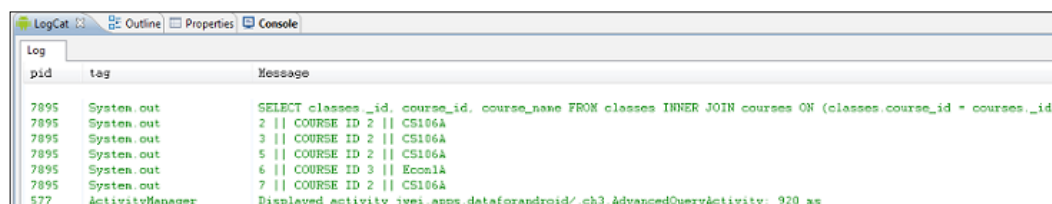
System.out.println(query);
c = sqdb.rawQuery(query, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(0);
    int colid2 = c.getColumnIndex(cols[1]);
    int colid3 = c.getColumnIndex(cols[2]);
    int rowId = c.getInt(colid);
    int courseId = c.getInt(colid2);
    String courseName = c.getString(colid3);
    System.out.println(rowId + " || COURSE ID " + courseId + "
    || " + courseName);
}
```

First, let me point out a couple of things specific to `JOIN` statements. In essence, a `JOIN` statement allows you to connect two tables based on some column values. For example, in our case, our schema was built so that we would have a table for classes and each class would be a mapping between the student ID and the course ID. But, let's say that we want to quickly know not just what the class mappings are but also the course's name for each mapping (that is, the name of the course and who is taking that class). Instead of having to return all the class mappings as well as the course listings (that is, asking for two tables back) and then manually doing these lookups, we can use an SQL `JOIN` statement to return a joint table.

Now, because when doing `JOIN` statements we are asking for multiple tables back, oftentimes when you ask for specific columns to return, you'll need to specify what table the column comes from. For instance, consider a situation where both tables have ID fields – in this case, simply asking for the ID column will cause an error, as it's ambiguous which table's ID column you really want. This is what we're doing when we initiate the strings `courseIdCol`, `classIdCol`, and `classCourseIdCol`, and the syntax is simply as follows:

```
table_name.column_name
```

Then in our `SQLiteQueryBuilder` class, we use the method `setTables()` to format our `JOIN` statement. Again, you can see the exact syntax that we used in the previous example, but the general format is first you specify the two tables that you want to join and then you tell the query what kind of `JOIN` you want (in our case, we want to use an `INNER JOIN`). Afterwards, you need to tell the query what two columns to perform the `JOIN` on, and again, in our case, we want to connect the two tables by the course ID, and so we specify the course ID column of our `Class` table and also specify the corresponding course ID column of our `Course` table. By doing this, the `JOIN` statement knows that for each class mapping, it should take the course ID and then go to the `Course` table and find that corresponding course ID and append that row of the table to the `Class` table. For an in-depth discussion on both the different kinds of `JOINS` as well as the syntax for each, I invite you to look at http://www.w3schools.com/sql/sql_join.asp and read through the documentation. The output for the previous `JOIN` statement is as follows:



pid	tag	Message
7895	System.out	SELECT classes_id, course_id, course_name FROM classes INNER JOIN courses ON (classes.course_id = courses._id)
7895	System.out	2 COURSE ID 2 CS106A
7895	System.out	3 COURSE ID 2 CS106A
7895	System.out	5 COURSE ID 2 CS106A
7895	System.out	6 COURSE ID 3 Econ1A
7895	System.out	7 COURSE ID 2 CS106A
577	ActivityManager	Displayed activity java.app.dataforandroid.ch3.AdvancedQueryActivity: 920 as

And so you can immediately see both the syntax of the query as well as the results.

SQL vs. Java performance comparisons

So just how powerful and efficient is the SQL language? In the previous two sections, we explored both basic and more advanced features of SQL – all of whose functionality (in theory) could be mimicked with just Java (that is, just do a bare-bones `SELECT` statement to get back the entire table and parse it with Java `if` statements, and so on). However, it's time to explore if there's an actual added advantage to filtering and manipulating our data on the SQLite end (as opposed to on the Java end), and if so, how much of an advantage it provides. And so, to start, we'll need a much bigger data set to better illustrate the improvements in performance.

First, we create a new table under a new schema which simply has a column for name, state, and income – think of this as a United States database with each family's name, the state they live in, and their family income. The table has 17,576 rows – still not a lot considering the magnitude of some real application tables – but hopefully this test table will illustrate some of these performance differences. Let's begin with the `WHERE` filter:

```
public class PerformanceActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        TestSchemaHelper sch = new TestSchemaHelper(this);
        SQLiteDatabase sqdb = sch.getWritableDatabase();

        // TEST WHERE FILTER PERFORMANCE //

        // SQL OPTIMIZED
        long start = System.nanoTime();
        String query = SQLiteQueryBuilder.buildQueryString(false,
            TestTable.TABLE_NAME, new String[] { TestTable.NAME },
            TestTable.INCOME + " > 500000", null, null, null, null);
        System.out.println(query);
        Cursor c = sqdb.rawQuery(query, null);
        int numRows = 0;
        while (c.moveToNext()) {
            int colid = c.getColumnIndex(TestTable.NAME);
            String name = c.getString(colid);
            numRows++;
        }
        System.out.println("RETRIEVED " + numRows);
    }
}
```

```

        System.out.println((System.nanoTime() - start) / 1000000 + "
        MILLISECONDS");
        c.close();

        // JAVA OPTIMIZED
        start = System.nanoTime();
        query = SQLiteQueryBuilder.buildQueryString(false,
        TestTable.TABLE_NAME, new String[] { TestTable.NAME,
        TestTable.INCOME }, null, null, null, null, null);
        System.out.println(query);
        c = sqdb.rawQuery(query, null);
        numRows = 0;
        while (c.moveToNext()) {
            int colid = c.getColumnIndex(TestTable.NAME);
            int colid2 = c.getColumnIndex(TestTable.INCOME);
            String name = c.getString(colid);
            int income = c.getInt(colid2);
            if (income > 500000) {
                numRows++;
            }
        }
        System.out.println("RETRIEVED " + numRows);
        System.out.println((System.nanoTime() - start) / 1000000 +
        " MILLISECONDS");
        c.close();
    }
}

```

On the SQLite side, we're simply using a WHERE filter which returns to us all families in our table that have a family income of over 500,000. On the Java side, we get back the entire table and loop through each row and use an if statement to perform the same filtering. We can verify that the outputted rows are the same, and at the same time look at the speeds of the two methods for comparison:

Time	pid	tag	Message
01-04 22:27:42.730	D 553	dalvikvm	GC freed 2 objects / 56 bytes in 120ns
01-04 22:27:42.810	I 6988	System.out	SELECT name FROM test_table WHERE income > 500000
01-04 22:27:44.390	D 6988	dalvikvm	GC freed 14518 objects / 495040 bytes in 92ns
01-04 22:27:44.620	I 6988	System.out	RETRIEVED 8745
01-04 22:27:44.630	I 6988	System.out	1818 MILLISECONDS
01-04 22:27:44.630	I 6988	System.out	SELECT name, income FROM test_table
01-04 22:27:48.693	D 6988	dalvikvm	GC freed 17177 objects / 567304 bytes in 96ns
01-04 22:27:50.820	D 6988	dalvikvm	GC freed 16401 objects / 525296 bytes in 80ns
01-04 22:27:51.820	I 6988	System.out	RETRIEVED 8745
01-04 22:27:51.820	I 6988	System.out	7193 MILLISECONDS

So we see that here there's almost a 5x boost in performance! Next, let's take a look at the performance boost gained when using the `GROUPBY` clause. On the SQLite side, we'll simply be doing a `GROUPBY` statement on the `states` column and will ask to count up how many families are from each state. Then, on the Java side, we'll ask for the whole table back and manually go through each row, using a standard `Map` object to keep track of each state and its respective count as follows:

```
// TEST GROUP BY PERFORMANCE //
```



```
// SQL OPTIMIZED
start = System.nanoTime();
String colName = "COUNT(" + TestTable.STATE + ")";
query = SQLiteQueryBuilder.buildQueryString(false, TestTable.
TABLE_NAME, new String[] { TestTable.STATE,
colName }, null, TestTable.STATE, null, null, null);
System.out.println(query);
c = sqdb.rawQuery(query, null);
while (c.moveToNext()) {
    int colid = c.getColumnIndex(StudentTable.STATE);
    int colid2 = c.getColumnIndex(colName);
    String state = c.getString(colid);
    int count = c.getInt(colid2);
    System.out.println("STATE " + state + " HAS COUNT " +
count);
}
System.out.println((System.nanoTime() - start) / 1000000 + "
MILLISECONDS");
c.close();
```



```
// JAVA OPTIMIZED
start = System.nanoTime();
query = SQLiteQueryBuilder.buildQueryString(false, TestTable.
TABLE_NAME, new String[] { TestTable.STATE },
null, null, null, null, null);
System.out.println(query);
c = sqdb.rawQuery(query, null);
Map<String, Integer> map = new HashMap<String, Integer>();
while (c.moveToNext()) {
    int colid = c.getColumnIndex(TestTable.STATE);
    String state = c.getString(colid);
    if (map.containsKey(state)) {
        int curValue = map.get(state);
        map.put(state, curValue + 1);
    } else {
```

```

        map.put(state, 1);
    }
}

for (String key : map.keySet()) {
    System.out.println("STATE " + key + " HAS COUNT " + map.
        get(key));
}

System.out.println((System.nanoTime() - start) / 1000000 + "
    MILLISECONDS");
c.close();

```

And let's see how well we did:

Time	pid	tag	Message
01-04 22:27:51.830	I 6988	System.out	SELECT state, COUNT(state) FROM test_table GROUP BY state
01-04 22:27:52.020	V 577	ActivityManager	Launch timeout has expired, giving up wake lock!
01-04 22:27:52.500	V 577	ActivityManager	Activity idle timeout for HistoryRecord(435b34a0 (jvei.apps.datafor
01-04 22:27:55.430	I 6988	System.out	STATE AR HAS COUNT 3460
01-04 22:27:55.430	I 6988	System.out	STATE CA HAS COUNT 3521
01-04 22:27:55.430	I 6988	System.out	STATE IL HAS COUNT 3534
01-04 22:27:55.430	I 6988	System.out	STATE NY HAS COUNT 3486
01-04 22:27:55.440	I 6988	System.out	STATE PA HAS COUNT 3568
01-04 22:27:55.440	I 6988	System.out	3608 MILLISECONDS
01-04 22:27:55.440	I 6988	System.out	SELECT state FROM test_table
01-04 22:27:57.639	D 619	dalvikvm	GC freed 1381 objects / 80920 bytes in 130ms
01-04 22:27:57.859	D 6988	dalvikvm	GC freed 19607 objects / 521240 bytes in 89ms
01-04 22:27:59.199	D 6988	dalvikvm	GC freed 21881 objects / 526520 bytes in 90ms
01-04 22:28:00.200	I 6988	System.out	STATE PA HAS COUNT 3568
01-04 22:28:00.200	I 6988	System.out	STATE AR HAS COUNT 3460
01-04 22:28:00.200	I 6988	System.out	STATE IL HAS COUNT 3534
01-04 22:28:00.200	I 6988	System.out	STATE NY HAS COUNT 3486
01-04 22:28:00.200	I 6988	System.out	STATE CA HAS COUNT 3521
01-04 22:28:00.210	I 6988	System.out	4763 MILLISECONDS

So we see that in this case, the performance boost was there but less noticeable, giving us a 33 percent boost in efficiency. It's important to note that these stated statistics are highly dependent on the schema and size of your tables, so take these numbers with a grain of salt. However, the goal of these little experiments is to just give us an idea of how these two methodologies compare. Lastly, let's take a look at how a built-in aggregate function like `avg()` in SQL compares with Java. The code for both methodologies is as follows:

```

// TEST AVERAGE PERFORMANCE //

// SQL OPTIMIZED
start = System.nanoTime();
colName = "AVG(" + TestTable.INCOME + ")";
query = SQLiteQueryBuilder.buildQueryString(false,
    TestTable.TABLE_NAME, new String[] { colName }, null, null,
    null, null, null);
System.out.println(query);
c = sqdb.rawQuery(query, null);
while (c.moveToNext()) {

```

```

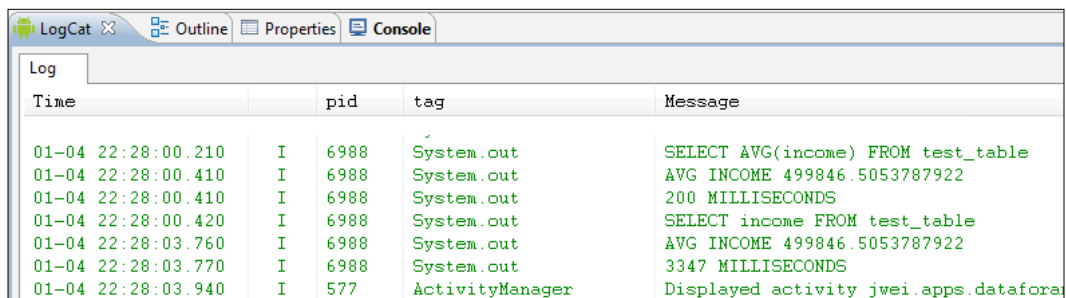
        int colid = c.getColumnIndex(colName);
        double avgGrade = c.getDouble(colid);
        System.out.println("AVG INCOME " + avgGrade);
    }
    System.out.println((System.nanoTime() - start) / 1000000 + "
    MILLISECONDS");
    c.close();

    // JAVA OPTIMIZED
    start = System.nanoTime();
    colName = TestTable.INCOME;
    query = SQLiteQueryBuilder.buildQueryString(false,
    TestTable.TABLE_NAME, new String[] { colName }, null, null,
    null, null, null);
    System.out.println(query);
    c = sqdb.rawQuery(query, null);
    double sumIncomes = 0.0;
    double numIncomes = 0.0;
    while (c.moveToNext()) {
        int colid = c.getColumnIndex(colName);
        int income = c.getInt(colid);
        sumIncomes += income;
        numIncomes++;
    }

    System.out.println("AVG INCOME " + sumIncomes / numIncomes);
    System.out.println((System.nanoTime() - start) / 1000000 + "
    MILLISECONDS");
    c.close();

```

And taking a quick look at what the output gives us:



Time	pid	tag	Message
01-04 22:28:00.210	I	6988	System.out SELECT AVG(income) FROM test_table
01-04 22:28:00.410	I	6988	System.out AVG INCOME 499846.5053787922
01-04 22:28:00.410	I	6988	System.out 200 MILLISECONDS
01-04 22:28:00.420	I	6988	System.out SELECT income FROM test_table
01-04 22:28:03.760	I	6988	System.out AVG INCOME 499846.5053787922
01-04 22:28:03.770	I	6988	System.out 3347 MILLISECONDS
01-04 22:28:03.940	I	577	ActivityManager Displayed activity jwei.apps.dataforai

And wow – enough said. The results for both methods are the same, but when using the SQL function you saw a 16x gain in performance.

Summary

In this chapter, we started by focusing on the Android OS and by looking at what types of query methods are at your disposal. We saw that there are three well-known ways to interact with the SQLite database, some more 'convenient' than the others, and some more flexible and powerful than the others.

However, we also saw that though each method had its pros and cons, all three query methods could ultimately perform the same kinds of queries, just using different sets of syntax or using different sets of parameters. This is when we moved away from the methods themselves and focused more on the query itself, starting with simple queries, which ranged from the most basic `SELECT` queries to more involved queries that allowed you to specify specific columns and rows. And later, we talked about more advanced queries, which ranged from `ORDERBY` and `GROUPBY` queries to the most complex and involved `JOIN` statements.

And lastly, being the curious and performance-minded programmers that we are, we spent the last section comparing the speeds of SQL and Java – implementing a variety of queries in both SQL and Java and then running them to look at the respective speeds. We saw that in each case, being able to embed your desired functionality into an SQL query granted you a performance boost when compared to executing that same functionality in Java (in one case it gave us as much as a 16x performance boost). And so, the moral of the story for this section is that when possible, find ways to manipulate your data on the SQL side as opposed to the Java side, as it will help you optimize speed as well as memory usage!

But before moving on, let's take a second to synthesize what we've learned so far. Earlier in *Chapter 2, Using a SQLite Database*, we learned about implementing SQLite database schemas in your Android application, and just now we learned about all the different features that are built into SQL which ultimately allow you to work with your data in extremely powerful, efficient ways. But now, what if you want to tap into existing data on the user's Android device? Each Android device contains a wealth of data, much of which is available for external applications to query, and so it's important to keep this in mind when developing your application. Furthermore, what if you want to expose your database and schemas to other applications? What if you're building a task list application and you want to allow other applications (perhaps calendar-based applications) to query for the user's existing tasks? All of these things are done through what's called a `ContentProvider`, and it's in the next two chapters that we flush out this extremely important class in Android.

4

Using Content Providers

We've accomplished a lot so far in this book! In just three chapters, we've looked at data storage mechanisms ranging from the simple, unassuming `SharedPreferences` class, to the powerful and complex `SQLite` database, equipped with a variety of query methods and classes that leverage the equally powerful language of SQL.

However, let's say that you've mastered the last three chapters and you've successfully built from scratch a database schema for your application that is now live in the market. Now, let's say you want to create a second application that extends the functionality of the first and requires access to your original application's database. Or perhaps you don't need to create a second application, but you simply want to better market your application by making available your database for external applications to access and integrate into their own.

Or, maybe you never even wanted to build your own database, but instead just wanted to tap into the wealth of data already existing on each Android device, and which is readily available for querying! In this chapter, we'll learn how to do all these things with the `ContentProvider` class, and at the end we'll spend some time brainstorming practical use cases of why you might benefit from exposing your database schema through a `ContentProvider`.

ContentProvider

Let's start with the question: What exactly *is* a `ContentProvider`? And why do I need to interact with this `ContentProvider`?

A `ContentProvider` is essentially an *interface* that sits between the developer and the database schema where the desired data sits. Why is this intermediary interface necessary? Consider the following (true) scenario:

In the Android OS, a user's contact list (this includes phone numbers, addresses, birthdays, and numerous other data fields pertaining to a contact) is stored in a fairly complex database schema on the user's device. Consider a scenario where as a developer, I'd like to query this schema for a user's contacts' phone numbers.

Think about how inconvenient it would be for me to have to learn the entire database's schema just to access one or two fields? Or how inconvenient it would be if every time Google updated the Android OS and tweaked the contact schema (and believe me, this has happened several times already), I had to relearn the schema and restructure my query subsequently?

It's for these reasons that such an intermediary exists—so that instead of having to interact directly with the schema, one only needs to query through the content provider. Now, on that note, each time Google updates its contact schema, they need to make sure they re-tweak their implementation of the `Contacts` content provider; otherwise our queries through the content provider may fail.

Said another way, much of this chapter and its implementation of the `ContentProvider` class is going to remind you of what we did earlier when writing convenience methods for our database. If you so choose to expose your data through a content provider, you will need to define how an external application can query your data, how an external application can insert new data or update existing data, and so on. These will all be methods that you'll need to override and implement.

But now let's be a little more discreet. There are many parts and pieces in implementing a content provider from start to finish, so to start, let's begin by laying out this section and looking at all of these pieces:

- Defining the data model (which is typically a SQLite database, which then extends the `ContentProvider` class)
- Defining its **Uniform Resource Identifier (URI)**
- Declaring the content provider in the Manifest file
- Implementing the abstract methods (`query()`, `insert()`, `update()`, `delete()`, `getType()`, and `onCreate()`) of `ContentProvider`

Now, let's start with defining the data model. Typically, the data model resembles that of a SQLite database (although it doesn't necessarily have to), which then simply extends the `ContentProvider` class. For my example, I've chosen to implement a pretty simple database schema consisting of just one table—a citizens table, meant to replicate a standard database that keeps track of a list of people who all have a unique ID (think social security ID), a name, a registered state, and in my case a reported income. Let's first define this `CitizensTable` class and its schema:

```

public class CitizenTable {
    public static final String TABLE_NAME = "citizen_table";
    /**
     * DEFINE THE TABLE
     */
    // ID COLUMN MUST LOOK LIKE THIS
    public static final String ID = "_id";
    public static final String NAME = "name";
    public static final String STATE = "state";
    public static final String INCOME = "income";
    /**
     * DEFINE THE CONTENT TYPE AND URI
     */
    // TO BE DISCUSSED LATER. . .
}

```

Pretty straightforward. Now let's create a class that extends the `SQLiteOpenHelper` class (just like we did earlier in the previous chapter), but this time we'll declare it as an inner class where the outer class extends the `ContentProvider` class:

```

public class CitizenContentProvider extends ContentProvider {
    private static final String DATABASE_NAME = "citizens.db";
    private static final int DATABASE_VERSION = 1;
    public static final String AUTHORITY =
        "jwei.apps.dataforandroid.ch4.CitizenContentProvider";
    // OVERRIDE AND IMPLEMENT OUR DATABASE SCHEMA
    private static class DatabaseHelper extends SQLiteOpenHelper{
        DatabaseHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            // CREATE INCOME TABLE
            db.execSQL("CREATE TABLE " + CitizenTable.TABLE_NAME +
                " (" + CitizenTable.ID + " INTEGER PRIMARY KEY
                AUTOINCREMENT," + CitizenTable.NAME + " TEXT," +
                CitizenTable.STATE + " TEXT," + CitizenTable.INCOME +
                " INTEGER);");
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion,
            int newVersion) {

```

```
        Log.w("LOG_TAG", "Upgrading database from version " +
            oldVersion + " to " + newVersion +
            ", which will destroy all old data");
        // KILL PREVIOUS TABLES IF UPGRADED
        db.execSQL("DROP TABLE IF EXISTS " +
            CitizenTable.TABLE_NAME);
        // CREATE NEW INSTANCE OF SCHEMA
        onCreate(db);
    }
}

private DatabaseHelper dbHelper;
// NOTE THE DIFFERENT METHODS THAT NEED TO BE IMPLEMENTED
@Override
public boolean onCreate() {
    // . . .
}
@Override
public int delete(Uri uri, String where, String[] whereArgs){
    // . . .
}
@Override
public String getType(Uri uri) {
    // . . .
}
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    // . . .
}
@Override
public Cursor query(Uri uri, String[] projection, String
    selection, String[] selectionArgs, String sortOrder) {
    // . . .
}
@Override
public int update(Uri uri, ContentValues values, String where,
    String[] whereArgs) {
    // . . .
}
}
```

You don't have to declare your SQLite database as an inner class—for me, it just makes the implementation a little easier and everything is nicely in one place. In any case, you'll notice that the implementation of the data model itself is exactly the same as before—override the `onCreate()` method and create your table, and then override the `onUpdate()` method and drop/recreate the table. In the skeleton we just saw, you'll also see the various methods that need to be implemented as a result of extending the `ContentProvider` class (this we will get into in the next section).

The only thing different about the code we just saw is the inclusion of the string:

```
public static final String AUTHORITY =  
    "jwei.apps.dataforandroid.ch4.CitizenContentProvider";
```

This authority is *what identifies the provider*—not necessarily the path. What I mean by this is that later on we'll see how you can define the entire *path* (this is known as the URI) to direct the query to the correct locations in your database schema.

In our content provider, we'll let developers query our database in one of two ways:

```
content://jwei.apps.dataforandroid.ch4.CitizenContentProvider/citizen  
  
content://jwei.apps.dataforandroid.ch4.CitizenContentProvider/  
citizen/#
```

Those are the two fully specified paths that we'll register in our content provider, and based on which path the developer requests, the content provider will know how to query our database. So what do these mean—notice that both start with the prefix `content://`, which is simply the standard prefix that tells the object this is a URI that points to a content provider (just as how `http://` tells the browser the path is pointing to a web page).

After the prefix we specify the authority so that the object knows which content provider to go to, and after that we have the suffixes `/citizen` and `/citizen/#`. The former we will simply define as the base query—the developer is just issuing a standard query and will pass any filters in the `query()` method. The second is for situations where the developer already knows the ID of the citizen (that is, the social security ID) and just wants to get a specific row of the table. Instead of forcing the developer to pass a `WHERE` filter with the ID, we can simplify things and allow the developer to specify the `WHERE` filter in the form of a path.

Now, in case all of this still sounds confusing, the most intuitive analogy to this would likely be: When you register an internet domain, you must specify a base URL, and once registered, the browser will know how to find the location of other files relative to this base URL. Likewise, in our case, we specify in the **Android manifest** (the motherboard of our application) that we want to expose a content provider and we define the path to it. Once registered, anytime a developer wants to reach our content provider, he/she must specify this *base* URI (that is, the authority), and furthermore he/she will need to specify what kind of query they are making by completing the path of the URI. For more on how the `ContentProvider` URI is defined, I invite you to check out:

<http://developer.android.com/guide/topics/providers/content-providers.html#urisum>

But for now, let's take a quick look at how you would declare your provider in the Android manifest file, and afterwards let's move on to the meat of the implementation, which is in overriding the abstract methods:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="jwei.apps.dataforandroid"
  android:versionCode="1"
  android:versionName="1.0">
  <application android:icon="@drawable/icon"
    android:label="@string/app_name">
    <provider
      android:name=
        "jwei.apps.dataforandroid.ch4.CitizenContentProvider"
      android:authorities=
        "jwei.apps.dataforandroid.ch4.CitizenContentProvider"/>
    </application>
  </manifest>
```

Again, pretty straightforward. All you need to do is define a name and authority for your content provider – in fact, the Manifest file will complain if you give an improper base URI as your authority, so as long as it compiles you know you're good to go! Now, let's move on to the more complex implementation of your content provider.

Implementing the query method

Now that we've built the data model, defined the table's authority and URI, and successfully declared it in our Android manifest file, it's time to write the bulk of the class and implement its six abstract methods. We'll begin with the `onCreate()` and `query()` methods:

```
public class CitizenContentProvider extends ContentProvider {
    private static final String DATABASE_NAME = "citizens.db";
    private static final int DATABASE_VERSION = 1;
    public static final String AUTHORITY =
        "jwei.apps.dataforandroid.ch4.CitizenContentProvider";
    private static final UriMatcher sUriMatcher;
    private static HashMap<String, String> projectionMap;

    // URI MATCH OF A GENERAL CITIZENS QUERY
    private static final int CITIZENS = 1;

    // URI MATCH OF A SPECIFIC CITIZEN QUERY
    private static final int SSID = 2;

    private static class DatabaseHelper extends SQLiteOpenHelper {
        // . . .
    }

    private DatabaseHelper dbHelper;
    @Override
    public boolean onCreate() {
        // HELPER DATABASE IS INITIALIZED
        dbHelper = new DatabaseHelper(getContext());
        return true;
    }

    @Override
    public int delete(Uri uri, String where, String[] whereArgs){
        // . . .
    }

    @Override
    public String getType(Uri uri) {
        // . . .
    }
}
```



```
@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    // . . .
}

@Override
public Cursor query(Uri uri, String[] projection,
    String selection, String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    qb.setTables(CitizenTable.TABLE_NAME);
    switch (sUriMatcher.match(uri)) {
        case CITIZENS:
            qb.setProjectionMap(projectionMap);
            break;
        case SSID:
            String ssid =
                uri.getPathSegments().
                get(CitizenTable.SSID_PATH_POSITION);
            qb.setProjectionMap(projectionMap);
            // FOR QUERYING BY SPECIFIC SSID
            qb.appendWhere(CitizenTable.ID + "=" + ssid);
            break;
        default:
            throw new IllegalArgumentException
                ("Unknown URI " + uri);
    }
    SQLiteDatabase db = dbHelper.getReadableDatabase();
    Cursor c = qb.query(db, projection, selection,
        selectionArgs, null, null, sortOrder);
    // REGISTERS NOTIFICATION LISTENER WITH GIVEN CURSOR
    // CURSOR KNOWS WHEN UNDERLYING DATA HAS CHANGED
    c.setNotificationUri(getContext().getContentResolver(),
        uri);
    return c;
}

@Override
public int update(Uri uri, ContentValues values, String where,
    String[] whereArgs) {
    // . . .
}

// INSTANTIATE AND SET STATIC VARIABLES
static {
```

```

        sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        sUriMatcher.addURI(AUTHORITY, "citizen", CITIZENS);
        sUriMatcher.addURI(AUTHORITY, "citizen/#", SSID);
        // PROJECTION MAP USED FOR ROW ALIAS
        projectionMap = new HashMap<String, String>();
        projectionMap.put(CitizenTable.ID, CitizenTable.ID);
        projectionMap.put(CitizenTable.NAME, CitizenTable.NAME);
        projectionMap.put(CitizenTable.STATE, CitizenTable.STATE);
        projectionMap.put(CitizenTable.INCOME,
            CitizenTable.INCOME);
    }
}

```

So let's just get the easy stuff out of the way first. You'll notice first off that after we define our SQLite database (by extending the `SQLiteOpenHelper` class), we declare a global `DatabaseHelper` variable and initialize it in our `onCreate()` method. The `onCreate()` method is called automatically after a request to open our particular content provider is made by an activity (through the use of a `ContentResolver` object, which we'll talk about later as well). Of course, any other initialization should go here, but in our case, all we want to do is initialize a connection to our database.

Once that's done, let's take a look at those static variables we've declared at the end. What the `projectionMap` does is it allows you to *alias your columns*. In most content providers, this mapping will seem a little meaningless, as you're simply telling the content provider to map your table's columns onto themselves (as we are doing in the implementation of the `onCreate()` and `query()` methods, which we just saw). However, there are certain instances where for more complex schemas (that is, ones with joint tables), being able to rename and alias your table's columns can make accessing your content provider's data much more intuitive.

Now, remember the two paths we talked about earlier (that is, `/citizen` and `/citizen/#`)? Well, all we're doing here is instantiating an `UriMatcher` object which allows us to define those paths through the method `addURI()`.

At a high level, what this method does is define a set of mappings—it's telling our `ContentProvider` class that any queries with path `/citizen` should be mapped to any behavior specified with the `CITIZENS` flag. Likewise, any queries with the path `/citizen/#` should be mapped to those behaviors specified by the `SSID` flag (these flags were both defined at the top of the class). Having this functionality can be useful for the developer as it allows him to efficiently query for a citizen if his/her ID is known ahead of time.

These flags then typically appear in `switch` statements, so now we'll focus our attention onto the `query()` method. It starts by initiating a `SQLiteQueryBuilder` class (which we spent a great deal of time looking at in our previous chapter), and from there it uses our `UriMatcher` object to match the passed-in URI. In other words, what the `UriMatcher` is doing is looking at the requested path and first figuring out if it's a valid path (if not, we throw an exception with error `unknown URI`). Once it sees that the developer has submitted a valid URI, it then returns that path's associated flag (that is, `CITIZENS` or `SSID` in our case), at which point we can use a `switch` statement to navigate to the proper functionality.

Once you understand what's happening at a high level, the rest should be pretty straightforward and familiar by now. If the user just submitted a general query (that is, with the `CITIZENS` flag), then all we need to do is define the projection map and the table name that will be queried. And again, if the user wants to go *directly* to a row in our table, then by specifying the social security ID in the path, we can parse that citizen out with the line:

```
String ssid =  
    uri.getPathSegments().get(CitizenTable.SSID_PATH_POSITION);
```

Don't worry too much about the `SSID_PATH_POSITION` variable—all we're doing here is taking the passed-in URI and breaking it into its path segments. Once we have the path segments, we're going to get the first one (and subsequently `SSID_PATH_POSITION` is set to 1 as we'll see soon), as in our example we only ever have one path segment passed in.

Now, once we have the desired social security ID that was passed into the query, all we need to do is append it to a `WHERE` filter and the rest is just stuff we've seen before—getting the readable database, and filling in the `query()` method of `SQLiteDatabase`.

The last thing I'll mention is that after the query has been successfully made and we get back our `Cursor` pointing at the data, since we are exposing our content provider to all external applications on the device, there is a chance that multiple applications may be accessing our database simultaneously, in which case our data is subject to change. Because of this, we tell our returned `Cursor` to *listen* for any changes that are made to its underlying data, so that when a change is made, the `Cursor` will know to update itself and subsequently any UI components that may use our `Cursor`.

Implementing the delete and update methods

Hopefully, everything makes sense at this point, so let's move on to the `delete()` and `update()` methods, which will look very similar to the `query()` method in structure:

```

public class CitizenContentProvider extends ContentProvider {
    private static final String DATABASE_NAME = "citizens.db";
    private static final int DATABASE_VERSION = 1;
    public static final String AUTHORITY =
        "jwei.apps.dataforandroid.ch4.CitizenContentProvider";
    private static final UriMatcher sUriMatcher;
    private static HashMap<String, String> projectionMap;

    // URI MATCH OF A GENERAL CITIZENS QUERY
    private static final int CITIZENS = 1;

    // URI MATCH OF A SPECIFIC CITIZEN QUERY
    private static final int SSID = 2;

    private static class DatabaseHelper extends SQLiteOpenHelper {
        // . . .
    }
    private DatabaseHelper dbHelper;
    @Override
    public boolean onCreate() {
        // HELPER DATABASE IS INITIALIZED
        dbHelper = new DatabaseHelper(getContext());
        return true;
    }

    @Override
    public int delete(Uri uri, String where, String[] whereArgs) {
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        int count;
        switch (sUriMatcher.match(uri)) {
            case CITIZENS:
                // PERFORM REGULAR DELETE
                count = db.delete(CitizenTable.TABLE_NAME, where,
                    whereArgs);
                break;
            case SSID:
                // FROM INCOMING URI GET SSID
                String ssid =
                    uri.getPathSegments().
                        get(CitizenTable.SSID_PATH_POSITION);
                // USER WANTS TO DELETE A SPECIFIC CITIZEN
                String finalWhere = CitizenTable.ID+"="+ssid;
                // IF USER SPECIFIES WHERE FILTER THEN APPEND
                if (where != null) {

```

```
        finalWhere = finalWhere + " AND " + where;
    }
    count = db.delete(CitizenTable.TABLE_NAME,
        finalWhere, whereArgs);
    break;
default:
    throw new IllegalArgumentException
        ("Unknown URI " + uri);
}
getContext().getContentResolver().notifyChange(uri, null);
return count;
}

@Override
public String getType(Uri uri) {
    // . . .
}

@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    // . . .
}

@Override
public Cursor query(Uri uri, String[] projection,
    String selection, String[] selectionArgs, String sortOrder) {
    // . . .
}

@Override
public int update(Uri uri, ContentValues values, String where,
    String[] whereArgs) {
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
        case CITIZENS:
            // GENERAL UPDATE ON ALL CITIZENS
            count = db.update(CitizenTable.TABLE_NAME, values,
                where, whereArgs);
            break;
        case SSID:
            // FROM INCOMING URI GET SSID
            String ssid =
```

```

        uri.getPathSegments().
        get(CitizenTable.SSID_PATH_POSITION);
// THE USER WANTS TO UPDATE A SPECIFIC CITIZEN
String finalWhere = CitizenTable.ID+"="+ssid;
if (where != null) {
    finalWhere = finalWhere + " AND " + where;
}
// PERFORM THE UPDATE ON THE SPECIFIC CITIZEN
count = db.update(CitizenTable.TABLE_NAME, values,
    finalWhere, whereArgs);
break;
default:
    throw new IllegalArgumentException
        ("Unknown URI " + uri);
}
getContext().getContentResolver().notifyChange(uri, null);
return count;
}
// INSTANTIATE AND SET STATIC VARIABLES
static {
    // . . .
}
}

```

And so we see that the logic behind these two statements very much follows that of the `query()` method. We see that in the `delete()` method, we first get our writable database (note that in this case we don't need the help of a `SQLiteQueryBuilder`, as we are deleting something and not querying for anything), and then we direct the passed-in URI to our `UriMatcher`. Once the `UriMatcher` validates the path, it then directs it to the appropriate flag, at which point we can vary the functionality accordingly.

In our case, any queries with the `CITIZEN` path specification just become a standard `delete()` statement, while those with the `SSID` path specification become a `delete()` statement with an additional `WHERE` filter on the `ID` column of the table. Again, the intuition here is that we are deleting a specific citizen from our database. Look at the following snippet of code:

```

String finalWhere = CitizenTable.ID+"="+ssid;
// IF USER SPECIFIES WHERE FILTER THEN APPEND
if (where != null) {
    finalWhere = finalWhere + " AND " + where;
}

```

Note how we're appending the ID filter onto whatever original `WHERE` filter the user may have specified. It's important to remember details like this in your implementation—namely, that the developer may have passed in additional arguments along with the ID in the path specification, so your final `WHERE` filter should take all of these into consideration. The only detail left is in the line:

```
getContext().getContentResolver().notifyChange(uri, null);
```

Here what we're doing is requesting for the `Context` and the `ContentResolver` that made this call, and notifying it that a change to its underlying data was successfully made. Why this is important will become clearer when we talk about how to bind `Cursors` to the UI, but for now consider a situation where in your activity, you display the rows of the data as a list. Naturally, every time something alters a row of the data in the underlying database, you'd want your list to reflect those changes, so this is why we need to notify those changes made at the end of our methods.

Now, I won't say much about the `update()` method as the logic is identical to that of the `delete()` method—the only difference is in the calls made by the writable `SQLite` database that you get. So, let's push onwards and finish our implementation with the `getType()` and `insert()` methods!

Implementing the insert and getType methods

It's time to implement our final two methods and complete our `ContentProvider` implementation. Let's take a look:

```
public class CitizenContentProvider extends ContentProvider {
    private static final String DATABASE_NAME = "citizens.db";
    private static final int DATABASE_VERSION = 1;
    public static final String AUTHORITY =
        "jwei.apps.dataforandroid.ch4.CitizenContentProvider";
    private static final UriMatcher sUriMatcher;
    private static HashMap<String, String> projectionMap;

    // URI MATCH OF A GENERAL CITIZENS QUERY
    private static final int CITIZENS = 1;

    // URI MATCH OF A SPECIFIC CITIZEN QUERY
    private static final int SSID = 2;

    private static class DatabaseHelper extends SQLiteOpenHelper {
        // . . .
    }
}
```

```

private DatabaseHelper dbHelper;
@Override
public boolean onCreate() {
    // . . .
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
    // . . .
}

@Override
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
        case CITIZENS:
            return CitizenTable.CONTENT_TYPE;
        case SSID:
            return CitizenTable.CONTENT_ITEM_TYPE;
        default:
            throw new IllegalArgumentException("Unknown URI "
                + uri);
    }
}

@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    // ONLY GENERAL CITIZENS URI IS ALLOWED FOR INSERTS
    // DOESN'T MAKE SENSE TO SPECIFY A SINGLE CITIZEN
    if (sUriMatcher.match(uri) != CITIZENS) { throw new
        IllegalArgumentException("Unknown URI " + uri); }
    // PACKAGE DESIRED VALUES AS A CONTENTVALUE OBJECT
    ContentValues values;
    if (initialValues != null) {
        values = new ContentValues(initialValues);
    } else {
        values = new ContentValues();
    }
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    long rowId = db.insert(CitizenTable.TABLE_NAME,
        CitizenTable.NAME, values);
    if (rowId > 0) {
        Uri citizenUri =
            ContentUris.withAppendedId(CitizenTable.CONTENT_URI,
                rowId);
    }
}

```



```
        // NOTIFY CONTEXT OF THE CHANGE
        getContext().getContentResolver().notifyChange(citizenUri,
            null);
        return citizenUri;
    }
    throw new SQLException("Failed to insert row into " + uri);
}

@Override
public Cursor query(Uri uri, String[] projection,
    String selection, String[] selectionArgs, String sortOrder) {
    // . . .
}
@Override
public int update(Uri uri, ContentValues values, String where,
    String[] whereArgs) {
    // . . .
}
// INSTITUTE AND SET STATIC VARIABLES
static {
    // . . .
}
}
```

First, let's tackle the `getType()` method. This method simply returns the **Multipurpose Internet Mail Extensions (MIME)** type of the data object requested for a given URI, which really just means you are giving each row (or rows) of your data a distinguishable data type. This then allows developers, if needed, the ability to identify whether or not a `Cursor` pointing to your table is indeed retrieving valid *citizen* objects. The rules behind specifying MIME types for your data are:

- `vnd.android.cursor.item/` for a single record
- `vnd.android.cursor.dir/` for multiple records

Subsequently, we'll define our MIME types in our `CitizenTable` class (which is also where we define our columns and schema):

```
public class CitizenTable {
    public static final String TABLE_NAME = "citizen_table";
    /**
     * DEFINE THE TABLE
     */
    // . . .
    /**
```

```

    * DEFINE THE CONTENT TYPE AND URI
    */

    // THE CONTENT URI TO OUR PROVIDER
    public static final Uri CONTENT_URI = Uri.parse("content://" +
        CitizenContentProvider.AUTHORITY + "/citizen");

    // MIME TYPE FOR GROUP OF CITIZENS
    public static final String CONTENT_TYPE =
        "vnd.android.cursor.dir/vnd.jwei512.citizen";

    // MIME TYPE FOR SINGLE CITIZEN
    public static final String CONTENT_ITEM_TYPE =
        "vnd.android.cursor.item/vnd.jwei512.citizen";

    // RELATIVE POSITION OF CITIZEN SSID IN URI
    public static final int SSID_PATH_POSITION = 1;
}

```

So now that we have our MIME types defined, the rest is simply passing the URI in the `UriMatcher` (again) and returning the corresponding MIME type.

And last but not least, we have our `insert()` method. This method is slightly different, but not significantly so. The only difference is that when inserting something, it doesn't make sense to pass a SSID URI path (think about it – if you're inserting a *new* citizen how could you possibly already have a desired social security ID to pass into the URI). So in this case, if a URI that *does not* have the `CITIZEN` path specification passed in, we throw an error. Otherwise, we proceed and simply retrieve our writable database and insert the values into our content provider (this we've seen before as well).

That's it! The goal is that after seeing the complete implementation, all the pieces tie together and you start to understand, at least intuitively, what is happening in our `ContentProvider` class. As long as this makes sense intuitively, the rest will follow when you actually program and implement the content provider yourself!

Now, before moving on to practical reasons for exposing your data through a content provider, let's take a quick look at how you would interact with a content provider (let's just use ours for now) and subsequently introduce the `ContentResolver` class, which we've seen come up a few times by now. This will seem quick for now, but no worries – soon we will devote an entire chapter on querying the most commonly used content provider: the `Contacts` content provider.

Interacting with a ContentProvider

At this point, we've successfully implemented our own content provider, which can now be read, queried, and updated (assuming the proper permissions are granted) by external applications! To interact with a content provider, the first step is to acquire from your `Context` the associated `ContentResolver`. This class behaves very much like a `SQLiteDatabase` class in the sense that it has your standard `insert()`, `query()`, `update()`, and `delete()` methods (in fact, the syntax and parameters for the two classes are extremely similar as well), but it's designed especially for interacting with content providers through URIs that are passed in by the developer.

Let's take a look at how you would instantiate a `ContentResolver` within an `Activity` class, and then insert and query for data using both path specifications:

```
public class ContentProviderActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        ContentResolver cr = getContentResolver();
        ContentValues contentValues = new ContentValues();
        contentValues.put(CitizenTable.NAME, "Jason Wei");
        contentValues.put(CitizenTable.STATE, "CA");
        contentValues.put(CitizenTable.INCOME, 100000);
        cr.insert(CitizenTable.CONTENT_URI, contentValues);

        contentValues = new ContentValues();
        contentValues.put(CitizenTable.NAME, "James Lee");
        contentValues.put(CitizenTable.STATE, "NY");
        contentValues.put(CitizenTable.INCOME, 120000);
        cr.insert(CitizenTable.CONTENT_URI, contentValues);

        contentValues = new ContentValues();
        contentValues.put(CitizenTable.NAME, "Daniel Lee");
        contentValues.put(CitizenTable.STATE, "NY");
        contentValues.put(CitizenTable.INCOME, 80000);
        cr.insert(CitizenTable.CONTENT_URI, contentValues);

        // QUERY TABLE FOR ALL COLUMNS AND ROWS
        Cursor c = cr.query(CitizenTable.CONTENT_URI, null, null,
            null, CitizenTable.INCOME + " ASC");
        // LET THE ACTIVITY MANAGE THE CURSOR
        startManagingCursor(c);
        int idCol = c.getColumnIndex(CitizenTable.ID);
```

```

int nameCol = c.getColumnIndex(CitizenTable.NAME);
int stateCol = c.getColumnIndex(CitizenTable.STATE);
int incomeCol = c.getColumnIndex(CitizenTable.INCOME);
while (c.moveToNext()) {
    int id = c.getInt(idCol);
    String name = c.getString(nameCol);
    String state = c.getString(stateCol);
    int income = c.getInt(incomeCol);
    System.out.println("RETRIEVED ||" + id + "||" + name +
        "||" + state + "||" + income);
}
System.out.println("-----");
// QUERY BY A SPECIFIC ID
Uri myC = Uri.withAppendedPath(CitizenTable.CONTENT_URI,
    "2");
Cursor c1 = cr.query(myC, null, null, null, null);
// LET THE ACTIVITY MANAGE THE CURSOR
startManagingCursor(c1);
while (c1.moveToNext()) {
    int id = c1.getInt(idCol);
    String name = c1.getString(nameCol);
    String state = c1.getString(stateCol);
    int income = c1.getInt(incomeCol);
    System.out.println("RETRIEVED ||" + id + "||" + name +
        "||" + state + "||" + income);
}
}
}

```

So what's going on here is we first insert three rows into our database, so that the citizen table now looks like:

ID	Name	State	Income
1	Jason Wei	CA	100000
2	James Lee	NY	120000
3	Daniel Lee	NY	80000

From here, we use our content resolver to make a general query of our table (that is, just passing in the basic URI path specification) in an order of increasing incomes. Then, we use our content resolver to make a specific query using the `SSID` path specification. To do this, we utilize the static method:

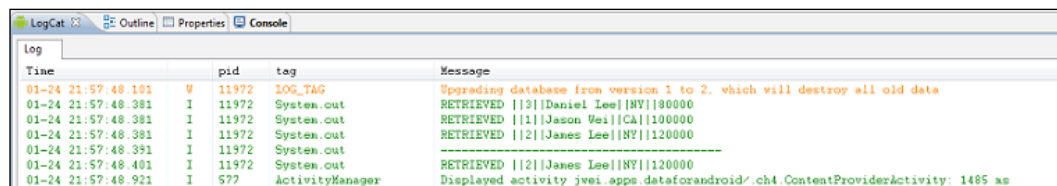
```
Uri myC = Uri.withAppendedPath(CitizenTable.CONTENT_URI, "2");
```

This transforms the base content URI from:

`content://jwei.apps.dataforandroid.ch4.CitizenContentProvider/citizen`
to the following:

`content://jwei.apps.dataforandroid.ch4.CitizenContentProvider/
citizen/2`

So, to validate our results, let's take a look at what was outputted:



Time	pid	tag	Message
01-24 21:57:48.101	W	LOG_TAG	Upgrading database from version 1 to 2, which will destroy all old data
01-24 21:57:48.381	I	System.out	RETRIEVED 3 Daniel Lee NY 80000
01-24 21:57:48.381	I	System.out	RETRIEVED 1 Jason Wei CA 100000
01-24 21:57:48.381	I	System.out	RETRIEVED 2 James Lee NY 120000
01-24 21:57:48.391	I	System.out	-----
01-24 21:57:48.401	I	System.out	RETRIEVED 2 James Lee NY 120000
01-24 21:57:48.921	I	ActivityManager	Displayed activity jwei.apps.dataforandroid/.ch4.ContentProviderActivity: 1485 as

From the previous screenshot, we can see that both queries indeed outputted the correct rows of data!

Now, the only remaining thing I'll say about the previous example (as most of the syntax and Cursor handling is identical to that of examples from previous chapters) is regarding the method `startManagingCursor()`. In earlier chapters, you'll notice that every time I open a Cursor through a query(), I have to make sure to close it at the end of the Activity; otherwise, the OS will throw out various hanging Cursor warnings. However, with the `startManagingCursor()` convenience method, the Activity will manage the life cycle of the Cursor for you — making sure to close it before the Activity destroys itself, and so on. In general, it's a good idea to allow the Activity to manage your Cursors for you.

Practical use cases

So, now that you know how to both implement and access a content provider, you might be scratching your head and thinking to yourself: *Why would I ever need to do this?*

What practical use cases are there for a content provider that would motivate you to go through the extra hassle of building a content provider instead of just extending a `SQLiteOpenHelper` and writing some convenience methods?

Well, one thing that is *unique* about the `ContentProvider` is that it allows you to expose your data to all external applications, and so we can start our brainstorming from there. Let's say you're running a small (or large) startup and you've developed an application that allows the user to look up restaurants and book reservations.

Now, sensibly, your application will most likely store these booked reservations in some kind of database, so that the user can see what reservations they made previously each time they open the application. But, say you expose your content provider and turn it into a *local* API (perhaps for some it's easiest to just think of a content provider as such) – in this case, other applications, perhaps a calendar application or a tasks list application, could develop some special functionality that allows them to *sync* their calendars and/or tasks with that user's restaurant reservations!

In this example, you have two applications, both with their own specific functionalities, leveraging the power of content providers to provide the user with a great experience (and happy users mean happy reviews for your application)!

Let's brainstorm one more example before we wrap up this chapter and move on to the next. One of the great things about the Android OS (and about Google in general) is the search functionality! As a result, within the Android OS, there's a native Quick Search application, which typically appears as a widget on the home screen of the device (see <http://developer.android.com/resources/articles/qsb.html> for more).

This Quick Search widget is especially cool because of how it allows you to search through *any and all* databases that declare themselves as searchable. And what prerequisites are there for making your database searchable? You guessed it – it has to be through a content provider. Again, it's only through exposing your data with a content provider that any application (whether native or third party) can read and access your database.

And so, say you are writing a texting application, and as a result you maintain a content provider that stores all of the most recent texts you've had with your friends. One neat feature you could add is to declare your content provider as searchable and then specify in your content provider what fields the search is to be done over (in this case, it would likely be the field containing the body of the text). Once you've done this, the user can quickly use the home screen's search widget and seamlessly maneuver through their texts with their friends!

At the end of the day, the principles and concepts behind the content provider are simple, and implementing is just half of the work – the other half is being creative and thinking of innovative and useful applications for your content provider.

Summary

In this chapter, we went into great detail about both what a `ContentProvider` is and how it is implemented, and as a result we saw a *lot* of code. However, conceptually, the `ContentProvider` is fairly simple—you first define an inner class that extends the `SQLiteOpenHelper`, and from there you specify how that SQLite database should be queried and/or modified, based on the *instructions* that are passed into each method. These instructions come in the form of URIs, and so in each method you're going to parse the different paths of the URI and perform the appropriate functionality.

We then quickly saw how you could interact with your new content provider (or any content provider, in fact) through the use of a `ContentResolver` which is obtained from the `Context` and then used to `query()`, `insert()`, `delete()`, or `update()` a corresponding content provider.

Lastly, we took some time to step away from the code and consider practical ways we could use a content provider. This is always an important exercise to do when developing an application, and is one of my goals for this book—to equip you with both the low-level implementation details of these techniques as well as the high-level motivations and use cases for them.

Now, earlier I mentioned that the Android OS is replete with pre-existing content providers that any developer is free to query and update. This is in fact true, and some of the more common content providers that are built into the system are the Media and Calendar content providers. However, by far the most important and most commonly used `ContentProvider` is the `Contacts` content provider—the database schema that is built into the OS and which houses the user's contacts list.

In the next chapter, we'll devote our entire attention into learning and understanding this `Contacts` content provider, its schema, and how to interact with it to accomplish standard queries and updates.

5

Querying the Contacts Table

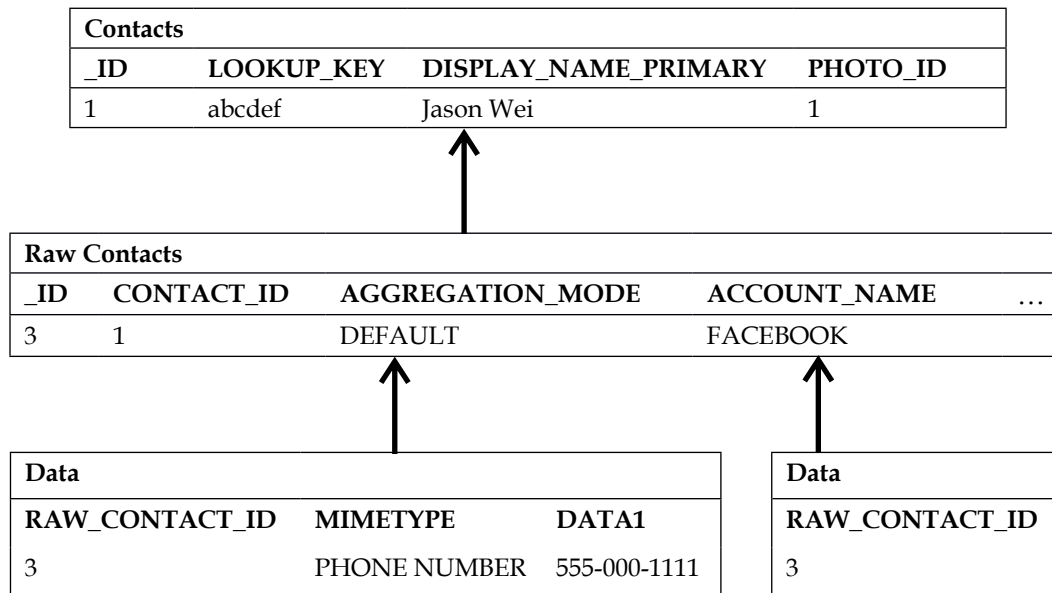
Earlier in this book, we looked at how we could build a SQLite database for our application by overriding the `SQLiteOpenHelper` class. Then, we extended our understanding of databases on Android by introducing the `ContentProvider` class, which allowed us to expose our SQLite databases to external applications, and more generally to the Android OS itself.

However, while knowing how to design and implement your own database is a powerful skill to have, knowing how to leverage existing data on the user's device can be just as beneficial. Oftentimes, this will mean querying existing content providers for various types of data, but one especially important content provider – and by far the most commonly queried content provider – is the `Contacts` content provider.

In this chapter, we'll start by exploring the structure of the `Contacts` content provider (that is, its schema) and then look at the various ways to query for contacts and their associated metadata.

Structure of the Contacts content provider

Understanding the schema of the `Contacts` content provider is half of the challenge. Because of the wealth of data that can potentially be associated with a contact, much work had to be done in designing a schema which would be both flexible and powerful enough to meet every user's needs. In the following table, I've sketched out how this schema is laid out, and from there we'll examine how the schema works at a high level, before diving into each table of the schema:



So here you have it – doesn't look too daunting right? Of course, the columns shown previously are just a subset of the actual columns in each table, but it should hopefully be enough to give you an idea of how these tables all work together. If you'd like to see all the columns in each table, I invite you to look at the following links:

<http://developer.android.com/reference/android/provider/ContactsContract.Contacts.html>

<http://developer.android.com/reference/android/provider/ContactsContract.RawContacts.html>

<http://developer.android.com/reference/android/provider/ContactsContract.Data.html>

Let's think about the schema from a high level first. At the top, we have the `Contacts` table. In previous versions of Android (API levels 4 and under), this was more or less all you had. It was just the typical, intuitive, `Contacts` table, which contained each contact's unique ID as well as their names, phone numbers, e-mails, and so on.

Then things got complicated. Suddenly, Android 2.0 (API levels 5 and up) came out and users were allowed to sync their contacts with Facebook, with Twitter, with Google, along with numerous other services. Does it still make sense to have just a simple `Contacts` table? Would each contact for each source be its own separate row? And how would we know which rows are actually referring to the same contact?

Because of this, Google had to develop a second layer of tables which reference the `Contacts` table – these tables are called `Raw Contacts`. Every contact the user has is an aggregation of raw contacts, where each raw contact represents a single contact from a specific source. So, say you had a friend and you've synced that contact with both Facebook and Twitter. This friend would then have two `Raw Contact` tables, one describing his/her metadata from Facebook and the other describing his/her metadata from Twitter. Both of these raw contacts would then both point to a single entry in the `Contacts` table.

But wait, whereas before each contact's metadata was more or less limited to a few phone numbers and a few e-mails, now there's an enormous amount of metadata available for each contact, thanks to social networking. So how would we store all this metadata? Each contact's latest status messages or latest tweets? Would we just have one enormous `Raw Contacts` table with thirty or so columns?

Preferably no – that's probably not a good use of memory, as that table would likely be fairly sparse. So instead, the team at Google decided to create a third layer of tables, known as the `Data` tables. These `Data` tables all reference a raw contact, which again, references a contact. And so that's essentially how a contact is described in the Android OS – a contact is an aggregation of raw contacts which are each specific to a source (that is, Facebook or Twitter) and each raw contact is an aggregation of separate data tables where each data table contains a certain type of data (that is, phone numbers, e-mails, status messages, and so on). That's the high-level picture of what's happening, and in the next section we'll look at how you actually query these tables for common fields, such as phone numbers and e-mails.

Now, there are many technical details that fully describe what's happening in the schema, but for now I'll end this section with a brief discussion of how this aggregation between raw contacts actually works.

The system automatically aggregates raw contacts, and so each time you create a new contact or sync a new account to an existing contact, that raw contact is created with aggregation mode set to `DEFAULT`, which tells the system to aggregate this raw contact with other raw contacts referencing the same contact. However, you can explicitly define what kind of aggregation you want for that raw contact and the options are as follows:

- `AGGREGATION_MODE_DEFAULT` – The default state, where automatic aggregation is allowed
- `AGGREGATION_MODE_DISABLED` – Automatic aggregation is not allowed and the raw contact will not be aggregated
- `AGGREGATION_MODE_SUSPENDED` – Automatic aggregation is deactivated, however, if the raw contact was previously aggregated, then it will remain aggregated

These are the three modes of aggregation, which you can update and adjust for each raw contact. As for how the aggregation is done, it's primarily done by matching names and/or nicknames, and if names are not present, then the match is attempted using phone numbers and e-mails.

By now you should have a decent understanding of what the `Contacts` content provider looks like, and so we'll move on to looking at some code!

Querying for Contacts

First, let's start with a simple query that targets the `Contacts` table and gives us back the contact IDs, each contact's name (remember this is an aggregated display name), as well as their lookup key. This lookup key is a relatively new concept to the `Contacts` content provider and is meant to be a more reliable way to reference `Contacts` than using the traditional row ID.

The reason for this is that row IDs tend to be unreliable, especially for a content provider like the `Contacts` content provider, which is likely to have numerous applications referencing, and potentially updating, it simultaneously. Say you try to reference a contact by its row ID, but earlier a different application on the user's device had made a change to the `Contacts` database so that either the contact at that row ID is now different, or perhaps now it is no longer there! Instead, the lookup key is a concatenation of the server side's identifiers for each raw contact (in other words, it is a function of the raw contact's metadata) and will be much more stable. But enough with the explanations, let's take a look at how a simple query might look:

```
public class ContactsQueryActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        /*
         * QUERY EXAMPLE
         */

        // FIRST QUERY FOR CONTACT LOOKUPS
        Cursor c = getContentResolver().query(
            ContactsContract.Contacts.CONTENT_URI,
            new String[] { ContactsContract.Contacts._ID,
                ContactsContract.Contacts.DISPLAY_NAME,
                ContactsContract.Contacts.LOOKUP_KEY },
            ContactsContract.Contacts.DISPLAY_NAME +
                " IS NOT NULL", null, null);
```

```

startManagingCursor(c);

int idCol = c.getColumnIndex(Contacts._ID);
int nameCol = c.getColumnIndex(Contacts.DISPLAY_NAME);
int lookCol = c.getColumnIndex(Contacts.LOOKUP_KEY);

// USE A MAP TO KEEP TRACK OF LOOKUP VALUES
Map<String, String> lookups = new HashMap<String, String>();

while (c.moveToNext()) {
    int id = c.getInt(idCol);
    String name = c.getString(nameCol);
    String lookup = c.getString(lookCol);
    lookups.put(name, lookup);
    System.out.println("GOT " + id + " // " + lookup +
        " // " + name + " FROM CONTACTS");
}
}
}

```

So, here we retrieve our content resolver as we did in the previous chapter, and pass in the `Contacts.CONTENT_URI`. We then iterate through the cursor and get the fields that we asked for in the projection array. Notice that I also use a `Map` to keep track of each contact's lookup key. In my case, I set the keys to be the contact's display name, but you could store the lookup keys and/or the contact IDs with any data structure that you like.

If you already know the lookup key of your contact (perhaps it had previously been cached somewhere), then you can use that lookup key to directly access the contact with the following snippet of code:

```

// ALTERNATIVELY - USE LOOKUP KEY LIKE THIS
Uri lookupUri = Uri.withAppendedPath(
    Contacts.CONTENT_LOOKUP_URI, lookups.get("Vicky Wei"));

Cursor c3 = getContentResolver().query(lookupUri,
    new String[] { Contacts.DISPLAY_NAME }, null, null, null);

if (c3.moveToFirst()) {
    int nameCol = c3.getColumnIndex(Contacts.DISPLAY_NAME);
    String displayName = c3.getString(nameCol);

    System.out.println("GOT NAME " + displayName +
        " FOR LOOKUP KEY " + lookups.get("Vicky Wei"));
}
c3.close();

```

So, here we append the `lookup` value to the URI itself – similar to how we earlier appended a row ID to the standard content URI to retrieve a single citizen. However, the problem with this method is that there tends to be some more overhead when trying to match by `lookup` key compared to the traditional match by row ID. In other words, you sacrifice some performance in speed in order to obtain better accuracy with your query. However, Android provides you with one more method that is meant to give you both increased accuracy and increased performance:

```
Uri lookupUri = getLookupUri(contactId, lookupKey)
```

This method allows you to first search for a contact by its contact ID – a much faster and still somewhat reliable method. However, in the event that a contact is not found with that contact ID, the system reverts to using the `lookup` key. In either case, as long as the contact exists, you're guaranteed to retrieve the correct `lookup` URI for that contact, but oftentimes using this method will give you a nice performance boost without sacrificing any accuracy.

Now that you have the contact IDs, the `lookup` keys, and their names, how would you query for more specific metadata – say their phone numbers or e-mails? Let's take a look at the following example, where I request a contact's phone number and phone type by filtering through their `lookup` key:

```
// THEN WITH LOOKUP KEYS - FIND SPECIFIC DATA FIELDS
Cursor c2 = getContentResolver().query(
    ContactsContract.Data.CONTENT_URI,
    new String[] { ContactsContract.CommonDataKinds.Phone.NUMBER, Phone.
    TYPE }, ContactsContract.Data.LOOKUP_KEY + "=?",
    new String[] { lookups.get("Vicky Wei") }, null);
startManagingCursor(c2);

int numberCol = c2.getColumnIndex(Phone.NUMBER);
int typeCol = c2.getColumnIndex(Phone.TYPE);
if (c2.moveToFirst()) {
    String number = c2.getString(numberCol);
    int type = c2.getInt(typeCol);
    String strType = "";
    switch (type) {
        case Phone.TYPE_HOME:
            strType = "HOME";
            break;
        case Phone.TYPE_MOBILE:
            strType = "MOBILE";
            break;
    }
}
```

```

        case Phone.TYPE_WORK:
            strType = "WORK";
            break;
        default:
            strType = "MOBILE";
            break;
    }
    System.out.println("GOT NUMBER " + number + " OF TYPE " +
        strType + " FOR VICKY WEI");
}

```

Notice that we leave some of the full package paths to the Phone and Data classes to again give you a glimpse at the hierarchical nature of the schema. Here, since we're targeting the Data tables now and not the Contact table, we pass in the corresponding Data CONTENT_URI. Then, in the projection parameter, we request the phone number as well as the phone type, and in the selection parameter I make sure I filter by the lookup key. Once I successfully make the query, we simply move the cursor (at this point there's only one number associated with Vicky; otherwise, we would use a while loop) and grab the fields again. Notice that we write a simple switch statement, which allows us to convert the PHONE_TYPE - returned as an integer - into a more user-friendly string.

And last but not least, let's take a look at how we could query the Raw Contacts table:

```

// NOW LOOK AT RAW CONTACT IDS
c = getContentResolver().query(
    ContactsContract.RawContacts.CONTENT_URI,
    new String[] { ContactsContract.RawContacts._ID, RawContacts.ACCOUNT_
        NAME, RawContacts.ACCOUNT_TYPE, RawContacts.CONTACT_ID }, null, null,
    null);
startManagingCursor(c);

int rawIdCol = c.getColumnIndex(RawContacts._ID);
int accNameCol = c.getColumnIndex(RawContacts.ACCOUNT_NAME);
int accTypeCol = c.getColumnIndex(RawContacts.ACCOUNT_TYPE);
int contactIdCol = c.getColumnIndex(RawContacts.CONTACT_ID);

while (c.moveToNext()) {
    int rawId = c.getInt(rawIdCol);
    String accName = c.getString(accNameCol);
    String accType = c.getString(accTypeCol);
    int contactId = c.getInt(contactIdCol);

    System.out.println("GOT " + rawId + " // " + accName +
        " // " + accType + " REFRENCING CONTACT " + contactId);
}

```

This is particularly useful if you want to look at a contact's metadata for a specific source (say you only care about what information Facebook has on that contact). Then you could potentially filter the `Raw Contacts` table by the `ACCOUNT_NAME` or `ACCOUNT_TYPE`, and once you have the raw contact IDs associated with that specific source, you could then query the `Data` tables for any metadata associated with those specific raw contact IDs!

Now, let's take a quick look at how to modify contact data – more specifically, how to insert and update contact data. Note that in order to successfully run these Activities, we'll need to request special permissions in the `Android Manifest` file. However, for now, let's continue to focus on the code, and we'll make sure to take a detour and cover all the permissions at the very end.

Modifying Contacts

The code for the following examples should again look very familiar. And like I said earlier, half of the challenge is just mastering the schema and understanding how each table interacts with the others (it helps to see the schema laid out like previously – otherwise it can be extremely confusing and may require browsing through a lot of verbose documentation). Let's say we want to insert a new phone number for a user. Which table's URI should we reference?

Well, it'd have to be one of the `Data` tables and we should probably pass in the `MIMETYPE` of the data so that the content provider knows exactly which of the `Data` tables to insert the new row in. In this case, we'll specify the phone content type and pass in a number and a number type. The only field we're missing is the ID – whose phone `Data` table should this new row go into? Well, recalling that each `Data` table points to a `Raw Contact` table, it would make sense to pass in the raw contact ID of the contact.

So we try to repeat this thought process for every insert, update, or delete that we have to make, and the code ends up looking like this:

```
public class ContactsQueryActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

```

/*
 * INSERT EXAMPLE
 */
ContentValues values = new ContentValues();
// IN THIS CASE - EACH RAW ID IS JUST THE CONTACT ID
values.put(ContactsContract.Data.RAW_CONTACT_ID, 2);
values.put(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE);
values.put(Phone.NUMBER, "555-987-1234");
values.put(Phone.TYPE, Phone.TYPE_WORK);
Uri contactUri = getContentResolver().insert(
    Data.CONTENT_URI, values);

Cursor c4 = getContentResolver().query(contactUri,
    new String[] { Phone.NUMBER, Phone.TYPE }, null, null, null);
startManagingCursor(c4);

// READ BACK THE ROW
if (c4.moveToFirst()) {
    String number = c4.getString(numberCol);
    int type = c4.getInt(typeCol);
    String strType = "";
    switch (type) {
        case Phone.TYPE_HOME:
            strType = "HOME";
            break;
        case Phone.TYPE_MOBILE:
            strType = "MOBILE";
            break;
        case Phone.TYPE_WORK:
            strType = "WORK";
            break;
        default:
            strType = "MOBILE";
            break;
    }
    System.out.println("GOT NUMBER " + number + " OF TYPE " +
        strType + " FOR VICKY WEI");
}
}
}

```


Here we use our content resolver along with a `ContentValues` object to do a standard insert. Once we insert it, we're returned the URI of that newly-inserted row, and so we simply run a query on that URI and read back the data that we just inserted, just as a sanity check that the insertion worked. I'll point this out in the screenshot which follows.

Now, the developers over at Google encourage another way to do an insertion, and this is through using batch insertions. This is another relatively new concept for the Android OS and is a variant on the traditional `ContentValues` class. By using batch operations, not only will you gain a considerable boost in performance when inserting multiple rows at once (saves you time from having to switch from the client side to the server side), but it will also ensure **atomicity** in your insertion. This is just a fancy database word meaning that either all of the rows will get inserted or none will, so that if an error occurs midway through your insertions, the system will make sure to roll back those previous insertions so that the consistency of the database is preserved.

The syntax for these batch insertions is shown as follows and is pretty intuitive:

```
// NOW INSERT USING BATCH OPERATIONS
ArrayList<ContentProviderOperation> ops =
    new ArrayList<ContentProviderOperation>();

ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
    .withValue(Data.RAW_CONTACT_ID, 3)
    .withValue(Data.MIMETYPE, Email.CONTENT_ITEM_TYPE)
    .withValue(Email.DATA, "daniel@stanford.edu")
    .withValue(Email.TYPE, Email.TYPE_WORK)
    .build());
try {
    getContentResolver().applyBatch
        (ContactsContract.AUTHORITY, ops);
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("ERROR: BATCH TRANSACTION FAILED");
}
```

To wrap up this chapter, we'll take a quick look at how you could use this new batch operation mechanism to update the e-mail of a contact:

```
/*
 * UPDATE EXAMPLE
 */

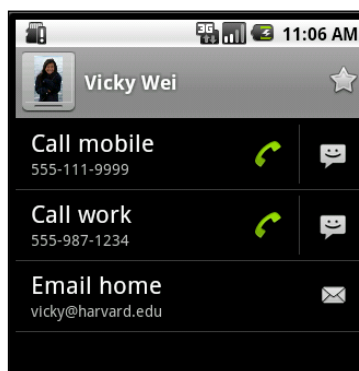
ops = new ArrayList<ContentProviderOperation>();
```

```
ops.add(ContentProviderOperation.newUpdate(Data.CONTENT_URI)
    .withSelection(Data.RAW_CONTACT_ID + "=?" AND " +
        Email.TYPE + "=?", new String[] { "7", String.valueOf(Email.
            TYPE_WORK) }).withValue(Email.DATA, "james@android.com").
    build());
try {
    getContentResolver().applyBatch(
        ContactsContract.AUTHORITY, ops);
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("ERROR: BATCH TRANSACTION FAILED");
    }
}
```

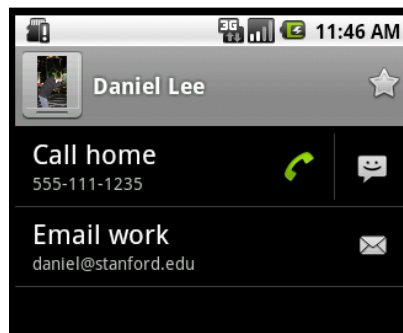
That's it! Here again, we think to ourselves that we'll likely need to specify both the raw contact ID so that the content provider knows whose Data table to update, as well as the MIME TYPE of the Data table so that the content provider knows which of the Data tables to update. As for the results from all the queries, insertions, and updates done in this section, see the following:

Log				
Time	pid	tag	Message	
02-04 11:44:47.308	I	19200	System.out	GOT 2 // 0n53392D3D59553139 // Vicky Wei FROM CONTACTS
02-04 11:44:47.318	I	19200	System.out	GOT 3 // 0n2F294339313F3F3131 // Daniel Lee FROM CONTACTS
02-04 11:44:47.318	I	19200	System.out	GOT 6 // 0n2D3739434D2943372943 // Chinsan Han FROM CONTACTS
02-04 11:44:47.318	I	19200	System.out	GOT 7 // 0n3B2941314D3F3131 // James Lee FROM CONTACTS
02-04 11:44:47.358	I	19200	System.out	GOT NUMBER 555-111-9999 OF TYPE MOBILE FOR VICKY VEI
02-04 11:44:47.398	I	19200	System.out	GOT NAME Vicky Wei FOR LOOKUP KEY 0n53392D3D59553139
02-04 11:44:47.438	I	19200	System.out	GOT 1 // null // null REFRENCING CONTACT 0
02-04 11:44:47.438	I	19200	System.out	GOT 2 // null // null REFRENCING CONTACT 2
02-04 11:44:47.448	I	19200	System.out	GOT 3 // null // null REFRENCING CONTACT 3
02-04 11:44:47.448	I	19200	System.out	GOT 4 // null // null REFRENCING CONTACT 0
02-04 11:44:47.448	I	19200	System.out	GOT 5 // null // null REFRENCING CONTACT 0
02-04 11:44:47.458	I	19200	System.out	GOT 6 // null // null REFRENCING CONTACT 6
02-04 11:44:47.458	I	19200	System.out	GOT 7 // null // null REFRENCING CONTACT 7
02-04 11:44:47.458	I	19200	System.out	GOT 8 // null // null REFRENCING CONTACT 0
02-04 11:44:47.458	I	19200	System.out	GOT 9 // null // null REFRENCING CONTACT 0
02-04 11:44:47.528	I	91	ContactAggregator	Contact aggregation: 1
02-04 11:44:47.588	I	91	ContactAggregator	Contact aggregation complete: 1. 65 ms per contact
02-04 11:44:47.688	I	19200	System.out	GOT NUMBER 555-987-1234 OF TYPE WORK FOR VICKY VEI
02-04 11:44:47.778	I	91	ContactAggregator	Contact aggregation: 1
02-04 11:44:47.868	I	91	ContactAggregator	Contact aggregation complete: 1. 91 ms per contact
02-04 11:44:48.028	I	91	ContactAggregator	Contact aggregation: 1
02-04 11:44:48.108	I	91	ContactAggregator	Contact aggregation complete: 1. 75 ms per contact

Here we first see all the contacts in my contact list along with their `lookup` keys, IDs, and display names. Then, we see the phone numbers we retrieved from Vicky, as well as the results of looking her up by her `lookup` key instead of her contact ID, and followed by our query of the `Raw Contacts` table. Notice that for account names and account types you see a bunch of null values, but this is simply a result of my running my code on the emulator. When you try running the code on a fully synced and live contact list, expect to see much more colorful results. Lastly, we just see some of the results from our insertions and updates and can further validate that our insertions/updates were successful by actually looking at the contacts in the contacts list as follows:



Here we see that we've successfully inserted a work number for contact Vicky, and then again for Daniel, we see the following:



So that he indeed now has a work e-mail with the correct e-mail that we specified. And so that's it! Hopefully, now you'll have a strong understanding of both the schema of the `Contacts` content provider as well as the general syntax for how you would construct a valid query or insertion. Remember to keep the schema in mind as you think through which fields to pass in and which tables you really want to be querying from.

Setting permissions

Now that we've mastered the `Contacts` content provider without declaring the proper permissions, you might encounter some rude force closes when trying to run the previous code. To protect the user's personal contact information from potentially malicious applications, the Android OS requires you to declare some read and write permissions in your applications in the `Android Manifest` file. To do this, all you need to do is add the following two lines to your manifest file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="jwei.apps.dataforandroid"
    android:versionCode="1"
    android:versionName="1.0">

    <application android:icon="@drawable/icon" android:label=
        "@string/app_name">

        <activity android:name=".ch5.ContactsQueryActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>
    <uses-sdk android:minSdkVersion="5" />

    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>

</manifest>
```

So basically, you just need to declare in your manifest that you want to be able to both read and write (that is, modify) contacts (or just declare one or the other according to what your application needs). This will then prompt the user before they download your application that your application requires these permissions, and as long as the user accepts them your application is all set to run!

Summary

In this chapter, we expanded upon our knowledge of content providers by mastering the most widely used content provider available to every application across every device – the `Contacts` content provider. We started off by taking a look at the schema of the `Contacts` content provider, which has grown increasingly complex as the amount of metadata associated with a given contact has soared thanks to various social networking sources. In order to solve this problem, the team at Google decided to switch up the schema by having a first-tier table simply known as the `Contacts` table, followed by a second-tier of tables known as the `Raw Contact` tables, and then by a third-tier of tables simply known as the `Data` tables. Each contact is then an aggregation of a group of raw contacts that are specific to a source (that is, Facebook or Twitter), and each raw contact is then an aggregation of a series of `Data` tables, each having its own type of data (that is, phone numbers or e-mails).

Afterwards, we looked at multiple ways to query the `Contacts` content provider as well as multiple ways to insert and update existing contacts in the content provider. This proved to be relatively straightforward code-wise (extremely similar to what we saw in previous chapters) and again, showed us how half the battle is just in understanding the schema and making sure we include all the proper fields.

Now, so far in this book, we've looked at ways to query your own as well as external databases, but each time we've relied on simple system print statements to actually see the results of our queries (by now I'm sure you're sick of seeing DDMS logs too). So the question becomes – now that I know how to actually build and query databases, how do I design Activities which allow me to bind this data to the UI for the user to see and interact with? This is what we'll focus on in the next chapter as we explore ways to bind and interact with our databases through the user interface.

6

Binding to the UI

We've covered a lot of ground over the previous five chapters – looking at lightweight forms of data storage (such as `SharedPreferences`) to more heavy-weight forms of data storage (such as `SQLite` databases). But for each data storage method and in each example that we've looked at – in order to actually see the results of our queries and our backend data manipulations, we had to rely on very simple system IO print commands.

More often than not though, as mobile developers, our applications will need to both aesthetically display the results of such data queries, as well as give users an intuitive interface to store and insert data.

In this chapter, we will focus on the former – on binding data to the user interface (UI) and will look specifically at various classes that will allow us to bind our data in the form of lists (the most common and intuitive way to display rows of data).

SimpleCursorAdapters and ListView

There are two major ways of retrieving data on Android, and each has its own class of `ListAdapters`, which will then know how to handle and bind the passed-in data. The first way of retrieving data is one that we're very familiar with already – through making queries and obtaining `Cursor` objects. The subclass of `ListAdapters` that wrap around `Cursors` is called `CursorAdapter`, and in the next section we'll focus on the `SimpleCursorAdapter`, which is the most straightforward instance of `CursorAdapter`.

As we already know, the `Cursor` points to a subtable of rows containing the results of our query. By iterating through this cursor, we are able to examine the fields of each row, and in previous chapters we've printed out the values of these fields in order to inspect the subtable that was returned. Now we would like to convert each row of the subtable into a corresponding row in our list. The first step in doing this is to set up a `ListActivity` (a variant of the more common `Activity` class).

As its name suggests, a `ListActivity` is simply a subclass of the `Activity` class which comes with methods that allow you to attach `ListAdapters`. The `ListActivity` class also allows you to inflate XML layouts, which contain list tags. In our example, we will use a very bare-bones XML layout (named `list.xml`) that only contains a `ListView` tag as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" >
    <ListView
        android:id="@android:id/list"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

This is the first step in setting up what's called a `ListView` in Android. Similar to how defining a `TextView` allows you to see a block of text in your `Activity`, defining a `ListView` will allow you to interact with a scrollable list of row objects in your `Activity`.

Intuitively, the next question in your mind should be: Where do I define how each row actually looks? Not only do you need to define the actual list object somewhere, but each row should have its own layout as well. So, to do this we create a separate `list_entry.xml` file in our `layouts` directory.

The example I'm about to use is the one that queries the `Contacts` content provider and returns a list containing each contact's name, phone number, and phone number type. Thus, each row of my list should contain three `TextViews`, one for each data field. Subsequently, my `list_entry.xml` file looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="10dip" >
    <TextView
        android:id="@+id/name_entry"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```

        android:textSize="28dip" />
    <TextView
        android:id="@+id/number_entry"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="16dip" />
    <TextView
        android:id="@+id/number_type_entry"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textColor="#DDD"
        android:textSize="14dip" />
</LinearLayout>

```

So we have a vertical `LinearLayout` which contains three `TextViews`, each with its own properly defined ID as well as its own aesthetic properties (that is, text size and text color).

In terms of set up – this is all we need! Now we just need to create the `ListActivity` itself, inflate the `list.xml` layout, and specify the adapter. To see how all this is done, let's take a look at the code before breaking it apart piece by piece:

```

public class SimpleContactsActivity extends ListActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list);

        // MAKE QUERY TO CONTACT CONTENTPROVIDER
        String[] projections = new String[] { Phone._ID,
        Phone.DISPLAY_NAME, Phone.NUMBER, Phone.TYPE };
        Cursor c = getContentResolver().query(Phone.CONTENT_URI,
        projections, null, null, null);
        startManagingCursor(c);

        // THE DESIRED COLUMNS TO BE BOUND
        String[] columns = new String[] { Phone.DISPLAY_NAME,
        Phone.NUMBER, Phone.TYPE };

        // THE XML DEFINED VIEWS FOR EACH FIELD TO BE BOUND TO
        int[] to = new int[] { R.id.name_entry, R.id.number_entry,
        R.id.number_type_entry };
    }
}

```



```
// CREATE ADAPTER WITH CURSOR POINTING TO DESIRED DATA
SimpleCursorAdapter cAdapter = new SimpleCursorAdapter(this,
R.layout.list_entry, c, columns, to);

// SET THIS ADAPTER AS YOUR LIST ACTIVITY'S ADAPTER
this.setAdapter(cAdapter);
}
}
```

So what's going on here? Well, the first part of the code you should recognize by now – we're simply making a query over the phone's contact list (specifically over the Contact content provider's Phone table) and asking for the contact's name, number, and number type.

Next, the `SimpleCursorAdapter` takes as two of its parameters, a string array and an integer array which represent a mapping between Cursor columns and XML layout views. In our case, this is as follows:

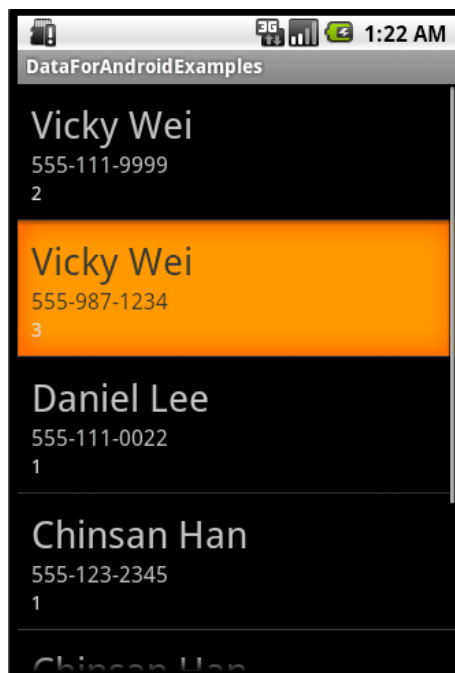
```
// THE DESIRED COLUMNS TO BE BOUND
String[] columns = new String[] { Phone.DISPLAY_NAME, Phone.NUMBER,
Phone.TYPE };
// THE XML DEFINED VIEWS FOR EACH FIELD TO BE BOUND TO
int[] to = new int[] { R.id.name_entry, R.id.number_entry, R.id.
number_type_entry };
```

This is so that the data in the `DISPLAY_NAME` column will get bound to the `TextView` with ID `name_entry`, and so on. Once we have these mappings defined, the next part is to just instantiate the `SimpleCursorAdapter`, which can be seen in this line:

```
// CREATE ADAPTER WITH CURSOR POINTING TO DESIRED DATA
SimpleCursorAdapter cAdapter = new SimpleCursorAdapter(this, R.layout.
list_entry, c, columns, to);
```

Now, the `SimpleCursorAdapter` takes five parameters – the first is the `Context`, which essentially tells the `CursorAdapter` which parent `Activity` it needs to inflate and bind the rows to. The next parameter is the ID of the `R` layout that you defined earlier, which will tell the `CursorAdapter` what each row should look like and, furthermore, where it can inflate the corresponding Views. Next, we pass in the `Cursor`, which tells the adapter what the underlying data actually is, and lastly, we pass in the mappings.

Hopefully, the previous code makes sense, and the parameters of `SimpleCursorAdapter` make sense as well. The result of this previous Activity can be seen in the following screenshot:



Everything looks good, except for these random integers floating around under the phone number. Why are there a bunch of 1s, 2s, and 3s at the bottom of each row where the types should be? Well, recall from the previous chapter that the phone number types are not returned as Strings but are instead returned as integers. From there through a simple `switch` statement, we can easily convert these integers into more descriptive Strings.

However, you'll quickly see that with our very simple, straightforward use of the built-in `SimpleCursorAdapter` class, there was nowhere for us to implement any "special" logic that would allow us to convert such returned integers to Strings. This is when overriding the `SimpleCursorAdapter` class becomes necessary, because only then can we have full control over how the Cursor's data is to be displayed in each row. And so, we move onwards to the next section where we see just that.

Custom CursorAdapters

In this section, we will expand upon the `SimpleCursorAdapter` and try to write our own `CursorAdapter` class, which will give us greater flexibility in terms of how the underlying data is to be displayed. The goal of our custom class will be simple – instead of having the phone number types being displayed as integers, let's find a way to display them as readable Strings.

Upon extending the `SimpleCursorAdapter` class, we'll need to override and implement the `newView()` method, and most importantly the `bindView()` method. Optionally, we can also customize our constructor, which depending on your implementation can be useful for caching and performance-enhancing reasons (we'll see an example of this later on).

Conceptually, what's happening here is that each time a new row is actually displayed on the Android device's screen, the `newView()` method gets called. This means that as the user scrolls through the Activity's list and new rows appear on the device's screen (for the first time), this `newView()` method will get called. And so, the functionality of this `newView()` should be kept relatively straightforward. In my implementation, this means that given the context, I make a request for the associated `LayoutInflater` class and use it to inflate the new row's layout (as defined in `list_entry.xml`).

The meat of the logic then occurs in the `bindView()` method. Once the `newView()` method is called and the actual layout of the row is initialized, the next method that gets called is the `bindView()` method. This method takes as parameters the new `View` object that was previously instantiated, as well as the `Cursor` that belongs to this adapter class. It's important to note that the `Cursor` that's passed in has already been moved to the correct index. In other words, the adapter is smart enough to pass you a `Cursor` that is pointing to the row of data corresponding to the row of your layout that you're creating! Now of course, it's hard to see and understand these methods without actually seeing the code side by side and so, before I go any further, let's take a quick look:

```
public class CustomContactsAdapter extends SimpleCursorAdapter {  
  
    private int layout;  
  
    public CustomContactsAdapter(Context context, int layout,  
        Cursor c, String[] from, int[] to) {  
        super(context, layout, c, from, to);  
    }  
}
```

```
        this.layout = layout;
    }

    @Override
    public View onCreateView(Context context, Cursor cursor,
        ViewGroup parent) {
        final LayoutInflater inflater = LayoutInflater.from(context);
        View v = inflater.inflate(layout, parent, false);
        return v;
    }

    @Override
    public void bindView(View v, Context context, Cursor c) {
        int nameCol = c.getColumnIndex(Phone.DISPLAY_NAME);
        int numCol = c.getColumnIndex(Phone.NUMBER);
        int typeCol = c.getColumnIndex(Phone.TYPE);

        String name = c.getString(nameCol);
        String number = c.getString(numCol);
        int type = c.getInt(typeCol);

        String numType = "";
        switch (type) {
            case Phone.TYPE_HOME:
                numType = "HOME";
                break;
            case Phone.TYPE_MOBILE:
                numType = "MOBILE";
                break;
            case Phone.TYPE_WORK:
                numType = "WORK";
                break;
            default:
                numType = "MOBILE";
                break;
        }

        // FIND THE VIEW AND SET THE NAME
        TextView name_text = (TextView) v.findViewById(
            R.id.name_entry);
        name_text.setText(name);
    }
}
```

```
        TextView number_text = (TextView) v.findViewById  
            (R.id.number_entry);  
        number_text.setText(number);  
  
        TextView type_text = (TextView) v.findViewById  
            (R.id.number_type_entry);  
        type_text.setText(numType);  
    }  
}
```

Again, you'll notice that the `newView()` method's implementation is pretty straightforward. You'll also notice that the Context being passed in is the same Context for each new row that is added – and so each time this method gets called, I'm actually requesting the same `LayoutInflater` object. Though it didn't make a noticeable difference in this case, little nuances like this (that is, not requesting the same resource continuously) are small ways in which you can optimize the performance of your lists. Here, by instantiating the `LayoutInflater` a single time in the constructor and reusing it each time, we can potentially save hundreds of unnecessary requests. Though this may seem like a very minor optimization, keep in mind that when it comes to lists, especially on mobile devices, users expect them to be extremely snappy and responsive. A list that lags is often a huge nuisance to users over time, and is frequently indicative of a poorly written application.

Now for the `bindView()` method. Again, the flow is that first `newView()` gets called and a new row is instantiated, and then `bindView()` gets called with this new row's layout view passed in. Here we have also passed a `Cursor` object, but it's important to note that the `Cursor` is actually pointing to the next row of data. In other words, the `Cursor` is not pointing to the first row of the queried subtable but instead is pointing to a single row and is being incremented accordingly behind the scenes. This is what I mean by the `CursorAdapter` class being a nice class to use because of how it handles the underlying `Cursor` for you as the list scrolls up and down.

As for the logic in our binding – it's pretty simple. Given the `Cursor`, we ask for the corresponding fields and their respective values, and since we're also passed the `View` object of that row, we just need to set the correct `String` value for each `TextView`. However, notice that here we have the flexibility to insert additional logic which allows us to handle the fact that the phone number's type is returned as an integer. So, naturally we include the switch statement here, and instead of setting the integer into the `type_text` `TextView`, we set the readable `String` value there!

Now, even though this is a pretty simple example, the goal of this exercise is to see how by extending the `SimpleCursorAdapter` class and implementing our own `CursorAdapter`, we can override the `bindView()` method and use the passed in `View` and `Cursor` objects to customize our row's display in any way that we want!

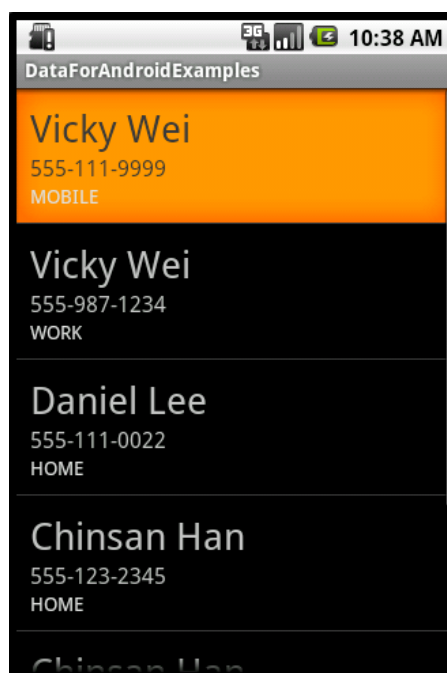
As for how you actually use your custom `CursorAdapter` in the previous `SimpleCursorAdapter` example, simply swap out the following line:

```
SimpleCursorAdapter cAdapter = new SimpleCursorAdapter(this, R.layout.  
list_entry, c, columns, to);
```

with the line:

```
CustomContactsAdapter cAdapter = new CustomContactsAdapter(this,  
R.layout.list_entry, c, columns, to);
```

And how does this all look in the end? Let's take a quick look:



Here we see that in each row, instead of simply showing the integer type of the phone number, we can see the actual readable `String` type as desired! Much nicer now.

BaseAdapters and Custom BaseAdapters

Earlier we mentioned that there were typically two ways to retrieve data – the first being in the form of a `Cursor` object and the second being in the form of a list of objects. In this section, we'll focus on this latter method of retrieving and handling data, and subsequently how to convert lists of objects into viewable rows of data.

So in what situations would we actually have a list of objects instead of a `Cursor`? Up until now, all of our focus has been on building up SQLite databases and content providers and in all cases we've been returned a `Cursor`. But, as we'll see in future chapters, oftentimes data storage isn't actually done on the mobile device side, but instead on external databases.

In these cases, retrieving data isn't as easy as just making SQLite queries, but instead needs to be done over the network through HTTP requests. Furthermore, once the data is obtained, it will likely be in some kind of String format (typically either XML or JSON – but more on this later), and instead of parsing this String for data and then inserting it into a SQLite database, typically you will simply convert each String into an object and store them in a standard list. To handle lists of objects, Android has a kind of `ListAdapter` known as the `BaseAdapter`, which we will override and dissect in this section.

Let's take a simple example where we have a list of contact objects (for simplicity, let's just call the class `ContactEntry`), which, like the previous examples, contain a name, phone number, and phone number type field. The code for this would simply be as follows:

```
public class ContactEntry {

    private String mName;

    private String mNumber;

    private String mType;

    public ContactEntry(String name, String number, int type) {
        mName = name;
        mNumber = number;
        String numType = "";
        switch (type) {
            case Phone.TYPE_HOME:
                numType = "HOME";
                break;
            case Phone.TYPE_MOBILE:
                numType = "MOBILE";
                break;
            case Phone.TYPE_WORK:
                numType = "WORK";
                break;
            default:
                numType = "MOBILE";
        }
    }
}
```

```

        break;
    }
    mType = numType;
}

public String getName() {
    return mName;
}

public String getNumber() {
    return mNumber;
}

public String getType() {
    return mType;
}
}

```

Here you'll notice that in the constructor of the `ContactEntry`, I convert the integer type directly into the readable `String` type. As for the implementation, we create our own `ContactBaseAdapter` class and extend the `BaseAdapter` class, allowing us to override the `getView()` method.

Conceptually, the `BaseAdapter` is very similar to the `CursorAdapter` except that instead of passing in and holding onto a `Cursor`, we pass in and hold onto a list of objects. This is simply done in the constructor of the `BaseAdapter`, at which point we store a private pointer to that list of objects and can optionally write a bunch of wrapper methods around that list (that is, `getCount()`, `getItem()`, and so on). And again, just as how the `CursorAdapter` class knows how to manage and iterate through the `Cursor`, the `BaseAdapter` class will know how to manage and iterate through the list of objects given.

The meat then is in the `getView()` method of the `BaseAdapter`. Notice how in the `CursorAdapter` class we had both a `newView()` method as well as a `bindView()` method. Here, our `getView()` method is designed to play the role of both – instantiating new views where the row was previously null, and binding data to old rows where the rows had previously been inflated. Let's take a quick look at the code and try to connect all these pieces again:

```

public class ContactBaseAdapter extends BaseAdapter {

    // REMEMBER CONTEXT SO THAT IT CAN BE USED TO INFLATE VIEWS
    private LayoutInflater mInflater;

```



```
// LIST OF CONTACTS
private List<ContactEntry> mItems = new ArrayList<ContactEntry>();

// CONSTRUCTOR OF THE CUSTOM BASE ADAPTER
public ContactBaseAdapter(Context context,
List<ContactEntry> items) {
    // HERE WE CACHE THE INFLATOR FOR EFFICIENCY
    mInflater = LayoutInflater.from(context);
    mItems = items;
}

public int getCount() {
    return mItems.size();
}

public Object getItem(int position) {
    return mItems.get(position);
}

public View getView(int position, View convertView,
ViewGroup parent) {
    ContactViewHolder holder;

    // IF VIEW IS NULL THEN WE NEED TO INSTANTIATE IT BY INFLATING
    IT - I.E. INITIATING THAT ROWS VIEW IN THE LIST
    if (convertView == null) {
        convertView = mInflater.inflate(R.layout.list_entry,
        null);

        holder = new ContactViewHolder();
        holder.name_entry = (TextView) convertView.findViewById(
        R.id.name_entry);
        holder.number_entry = (TextView) convertView.
        findViewById(R.id.number_entry);
        holder.type_entry = (TextView) convertView.findViewById(
        R.id.number_type_entry);

        convertView.setTag(holder);
    } else {
        // GET VIEW HOLDER BACK FOR FAST ACCESS TO FIELDS
        holder = (ContactViewHolder) convertView.getTag();
    }

    // EFFICIENTLY BIND DATA WITH HOLDER
    ContactEntry c = mItems.get(position);
```

```

        holder.name_entry.setText(c.getName());
        holder.number_entry.setText(c.getNumber());
        holder.type_entry.setText(c.getType());

        return convertView;
    }

    static class ContactViewHolder {
        TextView name_entry;

        TextView number_entry;

        TextView type_entry;
    }
}

```

First off, let's take a look at the constructor. Notice that I utilized the optimization mentioned earlier – namely, I instantiate the `LayoutInflater` just once in the constructor, because I know that the `Context` will remain the same throughout the `Activity`. This will give us a slight boost in performance when we actually run our `Activity`.

Now, let's see what's going on in this `getView()` method. The parameters for this method are the position (of the row), the row's view, and the parent view. The first thing we need to check is if the current row's view is null – this will be the case when the current row has not previously been instantiated, which in turn happens when the current row appears on the user's screen for the first time. If that's the case, then we instantiate and inflate this row's view. Otherwise, we know that we've already previously inflated this row's view, and simply need to update its fields.

Here, we also make use of a static `ContactViewHolder` class which acts as a cache. This method is recommended by the Android team over at Google (see <http://developer.android.com/resources/samples/ApiDemos/src/com/example/android/apis/view/List14.html> for details) and is meant to enhance the list's performance. The inflating of the view looks like the following:

```

if (convertView == null) {
    convertView = inflater.inflate(R.layout.list_entry, null);

    holder = new ContactViewHolder();
    holder.name_entry = (TextView) convertView.findViewById(
        R.id.name_entry);
    holder.number_entry = (TextView) convertView.
        findViewById(R.id.number_entry);
}

```

```
holder.type_entry = (TextView) convertView.findViewById  
(R.id.number_type_entry);  
  
convertView.setTag(holder);  
} else {  
    // GET VIEW HOLDER BACK FOR FAST ACCESS TO FIELDS  
    holder = (ContactViewHolder) convertView.getTag();  
}
```

Notice that when the view is null, the inflation of the view is pretty standard – use the `LayoutInflater` class and tell it which R layout to inflate. However, once the view has been inflated, we create an instance of the `ContactViewHolder` class and create pointers to each newly inflated view's `TextView` fields (in this case – though they could just as easily be `ImageViews`, and so on). Once the new `ContactViewHolder` class has been fully initiated, we associate it by setting it as the current row's tag (think of this as a view to holder mapping where the view is the key and the holder is the value).

If the view is not null, then we simply need to ask for the previously instantiated view's tag (again, you can think of this as requesting a key's value).

Once we have the corresponding `ContactViewHolder`, we can use the passed-in position to get the corresponding `ContactEntry` object in our list. From there, we know what contact the current row is referencing, and so we can dig out the name, number, and phone type, and then set them accordingly.

That's it! Let's take a look at how we would implement our `ContactBaseAdpater`:

```
public class CustomBaseAdapterActivity extends ListActivity {  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.list);  
  
        // MAKE QUERY TO CONTACT CONTENTPROVIDER  
        String[] projections = new String[] { Phone._ID,  
        Phone.DISPLAY_NAME, Phone.NUMBER, Phone.TYPE };  
        Cursor c = getContentResolver().query(Phone.CONTENT_URI,  
        projections, null, null, null);  
        startManagingCursor(c);  
  
        List<ContactEntry> contacts = new ArrayList<ContactEntry>();  
        while (c.moveToNext()) {  
            int nameCol = c.getColumnIndex(Phone.DISPLAY_NAME);
```

```

        int numCol = c.getColumnIndex(Phone.NUMBER);
        int typeCol = c.getColumnIndex(Phone.TYPE);

        String name = c.getString(nameCol);
        String number = c.getString(numCol);
        int type = c.getInt(typeCol);
        contacts.add(new ContactEntry(name, number, type));
    }

    // CREATE ADAPTER USING LIST OF CONTACT OBJECTS
    ContactBaseAdapter cAdapter = new ContactBaseAdapter(this,
    contacts);

    // SET THIS ADAPTER AS YOUR LIST ACTIVITY'S ADAPTER
    this.setAdapter(cAdapter);
}
}

```

For our purposes, you can ignore the first part, as we are literally querying the Contact content provider, taking the resulting `Cursor`, iterating through it, and creating a list of `ContactEntry` objects. Obviously this is silly, so assume that in your implementation you will directly be returned a list of objects. Once we have our list though, the call is simply:

```

// CREATE ADAPTER USING LIST OF CONTACT OBJECTS
ContactBaseAdapter cAdapter = new ContactBaseAdapter(this, contacts);

```

And the results of running this code look exactly like that of the second screenshot in our earlier example (as expected).

Now that we've taken a look at both `CursorAdapters` and `BaseAdapters` and how to implement each in code, let's take a step back and think about potential use cases for the two classes.

Handling list interactions

Now, one common feature of every `ListView` in Android is that the user should often be able to select a row in the list and expect some sort of added functionality. For instance, maybe you have a list of restaurants, and selecting a specific restaurant within the list brings you to a more detailed description page. This is again where the `ListActivity` class comes in handy, as one method we can override is the `onListItemClick()` method. This method takes several parameters, but one of the most important is the position parameter.

The full declaration of the method is as follows:

```
@Override
protected void onListItemClick(ListView l, View v, int position, long
id) { }
```

And once we have the position index, regardless of whether or not our underlying data is a `Cursor` or a list of objects, we can use this position index to retrieve the desired row/object. The code for the previous `CursorAdapter` example would look like the following:

```
@Override
protected void onListItemClick(ListView l, View v, int position,
long id) {
    super.onListItemClick(l, v, position, id);
    Cursor c = (Cursor) cAdapter.getItem(position);

    int nameCol = c.getColumnIndex(Phone.DISPLAY_NAME);
    int numCol = c.getColumnIndex(Phone.NUMBER);
    int typeCol = c.getColumnIndex(Phone.TYPE);

    String name = c.getString(nameCol);
    String number = c.getString(numCol);
    int type = c.getInt(typeCol);

    System.out.println("CLICKED ON " + name + " " + number + " "
+ type);
}
```

Similarly, the code for the `BaseAdapter` example would be as follows:

```
@Override
protected void onListItemClick(ListView l, View v, int position,
long id) {
    super.onListItemClick(l, v, position, id);
    ContactEntry c = contacts.get(position);

    String name = c.getName();
    String number = c.getNumber();
    String type = c.getType();

    System.out.println("CLICKED ON " + name + " " + number + " "
+ type);
}
```

Both are pretty similar and pretty self-explanatory. We simply retrieve the desired row/object using the position index, and then output the desired fields. Oftentimes, the developer might have a separate Activity where they would give the user more details on the object in the row they clicked (that is, the restaurant, the contact, and so on). This may require passing the ID (or some other identifier) of the row/object from the `ListActivity` to the new details Activity, and this is done through embedding fields into Intent objects – but more on this in the next chapter.

Comparing CursorAdapters and BaseAdapters

So under what typical scenarios would you find yourself using a `BaseAdapter` instead of a `CursorAdapter` and vice versa? We've already thought of a few instances previously, but let's take a little more time to brainstorm some use cases, just to get you even more comfortable with the two `ListAdapters` and when to switch between the two.

The general rule of thumb should be whenever your underlying data is returned as a `Cursor`, use a `CursorAdapter`, and whenever your data is returned or can be manipulated into a list of objects, use a `BaseAdapter`.

This means that for most network requests when the data is returned as one long String (again, getting a little ahead of myself but this String will typically be in either an XML or JSON format), it's best to simply parse the String and convert it into objects. These can then be stored in a list and passed into a custom `BaseAdapter`. This will often also be the case if you are calling an external API, in which case the data will typically come back as either XML or JSON. The exception then is when you want to cache the results.

Caching typically involves temporarily storing some data in a more local (or faster) area of memory (with CPU systems, this means storing data in RAM instead of on disk, and for mobile applications this means storing data locally instead of continuously requesting external data through a network). If you want to cache some of your network calls – whether it's for performance reasons or for offline access reasons – then the suggested flow is to make your network request, retrieve the formatted data String, parse the data String, and insert the data into a SQLite database (meant to mimic the external database). Then, since your data is already housed in a SQLite database, it's best (and easiest) to just make a quick query and get back a `Cursor`.

Now, what about a scenario where you have a static list of primitive objects, for instance Strings? This would often be the case if you had some kind of fixed table of contents where the user has to select from a pre-defined list of options. In that case, both a `BaseAdapter` and a `CursorAdapter` would be overkill, and instead you should opt to use a much simpler kind of Adapter known as the `ArrayAdapter`. I tried not to spend any time on this kind of `ListAdapter`, as it's extremely simple to use and conceptually it's extremely simple as well – if you have a static Array of Strings and you want to make a list out of them, just pass that Array into an `ArrayAdapter` and you're good to go.

However, this is all I will say on the `ArrayAdapter` and I invite you to read through the example found on the following site:

<http://developer.android.com/resources/tutorials/views/hello-listview.html>

Otherwise, just remember that for lightweight static data, use the `ArrayAdapter`, for dynamic object-oriented data, use the `BaseAdapter` and for locally stored subtable based data, use the `CursorAdapter`.

Summary

In this chapter, we finally shifted our focus away from the backend and towards the frontend – getting an in-depth look at ways we can bind our data to the user interface. Of course, users can interact with data in numerous ways, but by far the most common is through a `ListView`.

`ListView`s and `ListActivities` are convenient classes which allow us to bind `ListAdapters` to the Activity and subsequently to the list layouts, handling events such as when a user touches a row in the list. `ListAdapters` are then classes which take in the underlying data and handle the binding process for you – namely, that as your list scrolls up and down you don't need to keep track of the position in the list; all that is done for you behind the scenes. Instead, all you need to do is choose which `ListAdapter` to use depending on the type of underlying data you have, and specify how you want the binding to occur.

Equipped with these `ListAdapters`, we were able to recreate a stripped-down version of our contact list and, more importantly, were given a taste for all of the ways we could take our data and display it in interactive, beautiful ways.

We finished off the chapter thinking about the use cases between each subclass of `ListAdapters` (seeing in total three different subclasses, the `CursorAdapter`, the `BaseAdapter`, and lastly the `ArrayAdapter`) and again, the hope is to build intuition into both the backend and frontend application design process.

In the next chapter, we'll continue our brainstorming and try to pull together everything that we've seen – walking through a handful of practical examples and discussing ways we could design our backend and frontend to implement those examples.

7

Android Databases in Practice

In the previous chapter, we finally looked at ways we could bind our backend database to the user interface. At this point, we know about all the various local storage methods built into the Android OS (*Chapter 1, Storing Data on Android* and *Chapter 2, Using a SQLite Database*), most notably the SQLite database, as well as ways to take advantage of the SQLite language to execute powerful queries (*Chapter 3, SQLite Queries*). Furthermore, we know how to expose our custom SQLite databases to external applications through content providers (*Chapter 4, Using Content Providers*), as well as how to query pre-existing content providers such as the `Contacts` content provider (*Chapter 5, Querying the Contacts Table*).

And so at this point, we've already equipped ourselves with a lot of tools—enough to start building out full-fledged applications. However, before we do, let's pause and think.

Should we actually be relying on *local* SQLite databases? What if something happens to the user's phone, and their data gets erased? Or more importantly, should each user have to download the *entire* dataset, and store it locally on their phones? Keep in mind that a phone's memory is rather limited and is a fraction of what a desktop computer would have.

All these questions come into play when we start thinking about how we're going to design our application. Therefore, in this chapter, we'll start by looking at some practical use cases for having a localized SQLite database for your Android application, and then move to other, more typical application designs specifically for data-centric applications (if your application is going to be a game then likely this won't apply).

Local database use cases

So let's start with different ways one would likely see an Android application use a localized SQLite database. To clarify, what I mean when I say a *localized* SQLite database is one that solely exists on the phone's memory, and more specifically within the application's allocated memory, and is not backed up/supported by an external database. This is in contrast to an *external* database, which would exist on a server (or in the cloud), and would serve as either a backup to the localized database, or as a central database, from where all applications would request, insert, update, and delete data.

For our first example, consider a puzzle-based application that keeps track of all the user's high scores for each level. The high scores table would have fields such as the rank of that respective score (that is first, second, third, and so on), the name of the user who obtained that score, as well as the score itself. Let's go through each form of data storage, and think about whether or not it would be a sensible way to accomplish the task at hand:

- **SharedPreferences:** Could we use a Map-based class to accomplish this? I guess if we only needed one high scores table (as opposed to one per level) and that table only had a few rows, we could get away with using a simple Map. But this probably isn't a very natural use of the `SharedPreferences` class, and we could probably do much better with a different type of data storage—so let's pass on this one for now.
- **External SD cards:** As you probably recall, writing to SD cards is extremely useful for saving and backing up files. In theory though, we could probably save these tables in a file format—particularly by saving them in **Comma Separated Values (CSV)** files (think of these as spreadsheets). Then, we could just have one CSV file per level, and since a CSV file is structured like a spreadsheet, we can very easily read these files in and bind them to something like a `GridView`. Now, one of the nice things about saving stuff to an SD card is that your data is naturally backed up. For instance, if the user had to uninstall and reinstall your application for whatever reason, those CSV files would still exist and the data would be preserved. On the flip side though, if for whatever reason the user removed their SD card or tampered with their SD card, then it's possible that the data may be missing or corrupted. In any case, using CSV files and external SD cards isn't a terrible solution, but it may not be the most optimal or natural one.

- **SQLite databases:** Given that we're trying to save a series of tables, naturally we should think about using some kind of database schema. Now, depending on how many levels there are in our game (and subsequently how many tables we would need), we could design a database schema that has one separate table for each level, and for each level we could just point the `Cursor` to the correct table's URI. However, consider a scenario where we have 50 levels. In that case, it might seem a little silly to create 50 identical tables with 50 unique URIs. And so, what we might do is add an additional field to our table for *level*. Then, when we make our query, we could filter the table by the *level* column, and sort the remaining sub-table by rank. Using a SQLite database in this case would be especially slick because of how we could bind our resulting `Cursor` directly to the UI through a `ListView`. Now, what's the problem here? Well, if the user has to uninstall your application, then it's extremely likely that your SQLite database will get wiped out from the phone's memory.
- **External databases:** Using an external database in this case could potentially get very messy. Why? First, let's think about what our schema would have to look like. Potentially, we could have one giant table that contains fields for the device issuing the request (that is, the phone number or username of the device requesting the data), for the level being requested, and so on, and then just make queries that contain a bunch of filter clauses. Or, a nicer solution might be to have a table per level, and for each table include the additional field of which device that row belongs to. As you can see, in either case the schema is going to look a little messy, but for now let's stick with the latter schema. Say your game does moderately well and reaches 100,000 active installs. Furthermore, say your game has 50 levels and for each high scores table, you keep the top 10 scores. Not unreasonable for a semi-popular game, right? Well, under this scenario, suddenly your external database has 50 tables with a million rows per table, leaving you with a fairly large and memory-intensive database. Then, you have to take into account that each time the user requests to see a high scores table, he/she will need to issue an HTTP request to your external database in order to retrieve the corresponding table. This HTTP request will be several magnitudes slower than a simple SQLite query to your local database. So what's the plus side to all this work? This method will allow you to backup every user's high scores, independent of how many times they uninstall and reinstall your application, or how many times they change phones, and so on. Another nice feature is that once you have all of the data from all of your users, you could potentially create a global high scores table—allowing your users to see not just what the high scores were for their specific Android device, but what the all-time high scores were across all users who play your game!

And so, even in this scenario there are pros and cons for using a localized database versus an external database. The questions you'd need to ask yourself in this case are:

- How important is it that I backup the user's high scores?
- How likely/useful would it be to build a global high scores table?

If your intended game and audience is one that is extremely competitive and you believe users will get extremely upset if reinstalling your application/switching phones means losing their high scores history, then it might be wise to use an external database. However, my best guess is that very few mobile games will cause user's to become *that* competitive, in which case it would be significantly more practical to just have a simple localized database.

The conclusion? For a normal puzzle-based game with a simple high scores table, a localized database does the trick. The format of the data (that is, a table) makes this database a natural choice, and the assumption that users won't care about whether or not their high scores are preserved make implementing a localized database much more practical than an external database.

Let's consider one more example before we move on. Say you want to create an application that allows users to better find cafes and coffee shops. Perhaps you want to add features that allow the user to filter cafes and coffee shops by availability of space (too many times I find myself wandering into a nearby Starbucks just to find that all the tables are taken) or by availability of Wi-Fi. Not a bad application – but where would you find your initial cafe/coffee shop database?

Thankfully, you run into a couple of APIs from various services (that is, Yelp, Zagat, and so on) which allow you to query their databases, so the data source is no longer a problem. But now what? How would you design your Android application's backend? Let's walk through our options again:

- `SharedPreferences`: This time it is pretty easy to see why a method as simple and lightweight as a `SharedPreferences` class would not be appropriate. We'll pass on this one.

- **External SD cards:** So, like in our previous example, one possible way to use an external SD card is to store your data in CSV files (i.e. spreadsheet format) and then read and write to those files. And so, what we might do here is upon entering our application for the first time, we make a series of API calls to load our initial database of cafe/coffee shops. We then write our data into a CSV file and reference/update this CSV file going forward. So far so good. But what happens when we want to start filtering our data? Say the user only wants to see locations near him/her, or only wants to see locations that have free Wi-Fi. When we're dealing with CSV files, there doesn't exist a notion of *querying* this CSV file—a file is a file and our only solution would be to open a connection to the file, iterate through each row, and manually pick out the rows that we want. In this example, though it would be slow and burdensome, in theory we could implement our backend with this SD card solution. However, it's easy to see how once our schema becomes more complicated (requiring multiple tables instead of just one), not being able to execute efficient, complex queries would lead to an extremely poor design decision. And not to mention some of the issues mentioned previously with users removing SD cards, corrupting SD cards, and so on. Maybe it's best we stay away from SD cards in this situation.
- **SQLite databases:** With SQLite databases, again, it's a natural solution given the inherent table format of our data. We could very easily create a schema that has fields for name, location, Wi-Fi availability, and so on, and then write a slew of queries that would quickly filter our data accordingly. Additionally, with SQLite databases, we would benefit from the ease at which our data could be bound to the UI. However, what would the mechanics of our backend look like? Upon reaching the application for the first time, would we need to hit all APIs and download the entire dataset of all cafes/coffee shops across the entire nation? If we don't, then we run into the problem of what happens when the user is traveling, or wants to look up locations outside of their current city—most likely our only solution would be to call the APIs for every new location that is introduced. If we do download the entire data set at once, then depending on the number of cafes/coffee shops in the US, we could run into issues with memory and performance. In both cases, we need to methodically choose how we're going to sync and update our SQLite database with the newest information available through the APIs, an entirely different problem in and of itself.

- **External databases:** With an external database, we can also take advantage of the inherent table format of our data. And just like with localized databases, we can still execute quick queries to filter our data. We benefit from having a centralized database, ensuring that each time a user makes a request for a subset of data, it will be the most up-to-date data. Furthermore, since our database would exist in an external server, we don't require any additional memory on the application side, and we should also see a big performance gain, as making one request to one external database is much faster than making several requests to several APIs. Where we lose (compared to the SQLite database) is what happens when the user is making the same request *repeatedly*. For instance, say a user opens the search `Activity`, searches for his/her desired list of locations, waits a few seconds for the network request to come back, and then accidentally closes that `Activity`. If the user then re-opens the application and goes back to that `Activity`, he/she will need to make the *same* network request and wait another couple of seconds just to get the *same* results back. This can often be a huge nuisance for active users, and given the relatively short attention span of many mobile users, could be lethal to the success of your application.

Now that we've run through the list of data storage methods available to us, let's just quickly summarize some of the pros and cons of each. First off, in terms of pure *implementation*, the localized database and the external database were clear winners. Then, in terms of *memory consumption*, the external database was a better choice than the localized database because of how the entire dataset could exist outside of the application. In terms of *performance*, the external database was nice in the sense that instead of hitting multiple APIs, we only need to hit one database (our own). However, the localized database was nice in the sense that a user could maneuver in and out of the search `Activity` without having to make any additional network calls.

There's no clear winner here, but there is a way to *combine* the two methods to design a robust backend that addresses all of the previously discussed issues. This combined method uses an external database as the central storage unit, but then uses a localized database as a *cache* to improve performance. In the next section, let's hone in on what it means to use a localized SQLite database as a cache for an external database instead of a standalone database.

Databases as caches

So what exactly is a cache? A **cache** is often defined as a place in memory that stores duplicate data so that it can be served faster in the future. In our case, this is exactly what we're looking for.

In our previous example, we saw that by using an external database, we were able to improve upon memory consumption and, at times, performance, without compromising implementation. Additionally, we could naturally ensure that all users have the same data, and that data is the most up-to-date. The only time relying solely on an external database suffers is when you have users maneuvering around your application, having to make identical (or similar) network requests to your external database each time, and then having to repeatedly wait for those network requests to come back.

One solution is to use a cache and only have to make the network request *once*. Then, when the network request finishes, store a duplicate version of the returned data on a localized database, so that if the user makes the same (or similar) request, our system only needs to make a local query instead of a network query.

To help you better understand the low-level implementation, let's take a closer look at how this cache would work.

So the user lands on your search `Activity` and issues a request. Let's say the request is for all cafes and coffee shops within three miles from his/her location that also have free Wi-Fi. One design choice that you'll have to make is how much data should you cache in this case? Of course, you could issue the request with all of the user's desired filters, and only cache those results. But what if the user suddenly decides he/she doesn't care about having free Wi-Fi? Or if the user decides to relax their search criteria and wants to look for all shops within five miles instead?

While having a cache will definitely improve performance, the real gain comes from how often your cache is hit. For those who are familiar with designing caches, the trade-off comes from the frequency at which your cache is hit versus the size of your cache. In other words, in an extreme case, if you designed your cache to contain your *entire* data set, then obviously every request would be a cache hit, and so your cache would be extremely effective in that sense. However, the fact that you have your entire data set stored in memory is sub-optimal (and oftentimes impossible depending on the size of your database), and so your cache would fail on that front. Trying to find a nice blend of the two is the goal, and so in this situation, instead of only requesting for locations within three miles that have free Wi-Fi, why not try requesting for all locations within five miles and exclude the Wi-Fi filter completely?

By caching this request instead, now when the user decides to relax his/her search conditions from three miles to five miles (or downwards to two miles), you'll already have all the results; so instead of issuing another network request you can simply filter your cache for the desired subset of data. Similarly, if your user wants to remove the Wi-Fi filter, you can quickly query your cache for this data, this time with the *Wi-Fi only* filter removed. In both cases, the user hits your cache and saves you from making a time-consuming network request.

The last leg in designing your caching system would just be determining how often to refresh your cache. Never refreshing your cache is sub-optimal, as it will only consume more memory over time with each new request you cache, and, furthermore, you'll run into the problem of having out-of-date data. For instance, say your user makes a cafe/coffee shop request for their hometown and you cache this result. However, your caching system is one that never refreshes the cache. A lot can happen in a year, and a year later when the user pulls out your application again and makes the same cafe/coffee shop request, he/she will hit the cache and pull the old data instead of making a fresh request.

On the flip side, by allowing your cache to refresh too frequently, you'll decrease your cache hit frequency and will end up having to make more network requests than desired. And so we again have an optimization problem where we wish to maximize the number of cache hits, while minimizing the amount of memory consumption needed, and also minimizing the frequency at which we pull stale data.

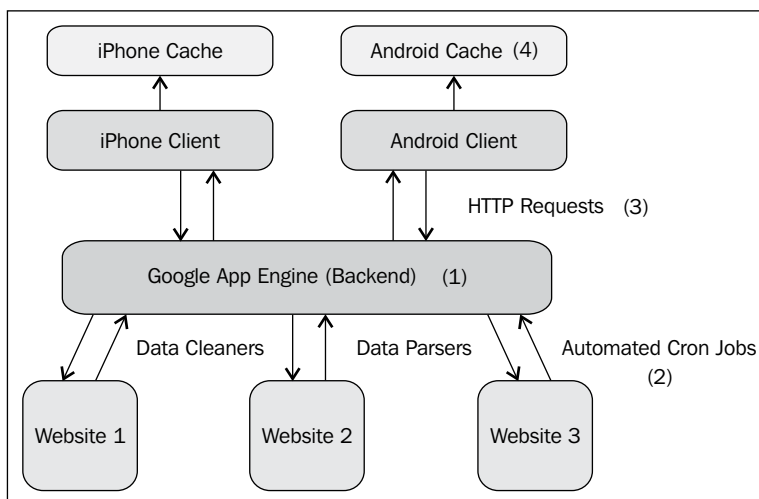
This simplified optimization problem sits at the heart of every caching system, and is the one you need to keep in mind when using a localized database to cache your external database network requests. While there is still much that could be discussed regarding caches, the goal of this section (and of this whole chapter in general) was to stir your thought process, and introduce you to one of the many uses for localized databases, and how they can be used in conjunction with external databases.

In the next section, I'll discuss how a typical data-centric application will look as well as map out the typical flow of data. Again, what I mean by *data-centric* applications are those whose primary functions involve displaying/interacting with some form of data. This could include everything from social networking applications where users can read/write messages to each other (the data in this case includes messages, events, photos — anything that can be shared), to food and dining applications where users can load details of nearby restaurants. This would typically exclude many game-based applications, though even game-based applications will at times need to adopt some kind of external database (for instance, the global high scores table we discussed earlier). With that, let's shift our focus again and start thinking about mobile applications from a more holistic perspective — as extensions of external databases and external applications, rather than simple standalone applications.

Typical application design

Up until now, we've only talked about and toyed with ideas regarding backend application designs. We first thought about the pros and cons of using a completely localized backend versus a completely external backend, and then thought about ways of using both in applications to try and get the best of both worlds. One of the ways we could do this was through using a cache, and in designing the cache alone, we saw that there were a slew of design decisions that had to be made.

Whether you realized this or not, this whole time you've been analyzing the pros and cons of different backend designs for different applications, and now we're ready to focus on a very general design that is extremely practical and is frequently used in data-centric mobile applications. But enough with words, let's put a picture to our design:



So what exactly is going on here? Let's break this down:

1. First, we have our external, centralized database. This is the heart of our backend. All applications (whether web or mobile) will reference this database, and in this way, we can ensure that all data across all mobile devices will be synced and up-to-date. Furthermore, in this design our application is no longer platform specific. In other words, one could easily create an application that works across all mobile devices, both Android and iOS, since all devices are pointing at the same database.

2. The external database also separates the client side (that is, the mobile applications and the web applications) from the data gathering/parsing/cleaning side. Here, in the latter, we have all of the processes meant to go and collect, parse, and clean the backend's data. This might involve periodically hitting APIs (assuming the API allows you to store copies of its data), scraping websites (to be discussed later), or, in some cases, manually inserting new data. Then, once the data comes in, it typically needs to be parsed and cleaned to fit the specifications of your database. Furthermore, this whole process of gathering and cleaning data can itself be automated with the use of CRON jobs (discussed in *Chapter 9, Collecting and Storing Data*). So by setting up your application in this way, you are able to conceal all of this behind the scenes data mining from your users.
3. On the flip side, you have the web applications and the mobile applications continuously making requests to your external database. These requests will typically be in the form of HTTP GET and POST requests (getting data versus inserting/ updating data) and will return results in either XML or JSON format. Again, because these are just standard HTTP network requests, it is independent of the platform making the request, so you can easily port applications from one platform to another.
4. Lastly, we have the cache, which is a temporary, localized subset of our external database, and which exists on the mobile/web application side. As discussed earlier, these caches are designed to increase performance of the application by saving us from having to make duplicate network requests.

And there you have it. Again, for the moment this is still very high level, but we've already seen and discussed the components related to part four of our design and in the upcoming chapters, we'll look at the first three parts as well.

Summary

Even though we didn't look at any code in this chapter, we still managed to accomplish a lot. We started the chapter by identifying two very realistic needs (a simple high scores table and then a location/venues database), and worked through the thought process behind how you would choose an appropriate storage method.

We saw that for something as simple as a high scores table, a localized SQLite database was both effective and simple to implement. The only real con to this approach was not being able to show global high scores tables, but for most games, this is but a minor feature. However, for our cafe/coffee shop application, we saw that a localized SQLite database was much less effective than having a centralized external database, with the only con for the external database solution being that performance would suffer if repeated, unnecessary network calls were frequently made.

To address this issue, we turned to caching as a solution—using both external and local databases and trying to leverage the pros of each method. However, to build an effective cache requires making several design decisions in order to optimize the cache hit frequency, while minimizing memory consumption and stale data.

Lastly, we ended the chapter by taking not just a step away from the code, but a step away from the Android application itself, and tried to look at our application from a more holistic view. We looked at what a typical data-centric application would look like, and broke the circulation of data down into four parts. Up to this point, we've already covered enough to be able to implement part four of the design (the local cache), and we'll now devote a chapter to each of the remaining three parts. By the end of this book, the goal is for you to be able to confidently design and implement a full-scaled data-centric application.

8

Exploring External Databases

In the previous chapter we introduced the notion of moving away from completely localized databases that exist solely on the Android client side, and towards utilizing an external database that could help us in several ways throughout the development process.

We saw how by using an external database, we were able to improve memory usage in our Android applications (namely, by not having to store extremely large database files) without sacrificing too much performance by using caches. Furthermore, we saw how using an external database allowed us to back up user data (in case a user switches phones or uninstalls your application), prevent users from seeing stale data (since all data exists in one central location), as well as potentially see other user's data (remember the global high scores example).

Using external databases that your application can communicate with over a network will make you a much more versatile application developer and will give you the tools to create fully scalable data-centric applications.

Different external databases

So what kinds of external databases are out there anyways? Just like how Android, iOS, Palm, and so on, are all examples of operating systems which allow you to develop mobile applications, there are several easily accessible platforms out there which allow you to host and develop external databases.

One such "platform" is just setting up a traditional dedicated server with database capabilities. For instance, a popular example of this would be having a dedicated computer hosting an **Apache Tomcat** server that's connected to a **MySQL** database. I won't go into the details of how you would set up this kind of server-database connection (primarily because you can do it in any number of ways), but instead let's just think about high-level concepts and then move on to a simple pros and cons list.

At a high level, the Apache Tomcat server acts as an intermediary that handles all incoming and outgoing HTTP requests (that is, network requests). The server itself listens for all these incoming requests, and upon receiving one, has code that tells it how to handle the request and subsequently what to return as a response. The code that handles the request and returns a response is often known as the **HTTP servlet**, and in upcoming chapters we'll actually implement a few of these servlets to give you a better idea for how they work.

Moving on though, the Apache Tomcat server is also connected to a MySQL database through a **Java Database Connectivity** driver (**JDBC**). Once configured, this will allow us to handle incoming HTTP requests, which then tell the server to issue a query to the MySQL database. Once the MySQL database retrieves the query, it will execute it and return the desired data, ultimately to be sent back to the original requester.

Using this kind of a model, the pros are that it's fully customizable and that you have full control over how each part is implemented. However, this can be a double-edged sword and can be a good or bad thing depending on who is handling the server and the database. Focusing on the database portion, because it is fully customizable, we have complete control over what **database management system (DBMS)** we want to use and furthermore what our database schema should look like for our given database management system. Throughout the application development process, we can even elect to switch our DBMS or alter our schema if we felt it was necessary – for instance, if we needed a more scalable DBMS.

And this is where the problem lies. Though MySQL is by far the world's most used DBMS and in most cases does a great job, it's not designed to be extremely scalable. Thus, for large, data-heavy applications, using MySQL would be a sub optimal design decision. And going back to my original point that using a fully customizable server and database can be a double-edged sword, one can easily see how flexibility and responsibility go hand in hand in this case. As we gain more flexibility in the design/implementation of our system, we simultaneously have more responsibility when it comes to making intelligent design decisions – otherwise, our application's performance may deteriorate quickly (that is, imagine if all of Google's data was hosted on a single computer – what a nightmare).

Other cons are that these systems typically require a higher cost initially, as we need to actually buy computers/servers. In addition, because these computers/servers are prone to failure, we'll have to manage them regularly to make sure they don't crash. Because of their flexibility, many companies and startups opt for this model, though many end up hiring specialists dedicated to maintaining these servers as well as backend developers dedicated to building out these servers and databases.

Recently, though, the idea of cloud computing has become increasingly popular, and here I'll introduce two such platforms. The first is **Amazon's Web Services (AWS)**, which provides a suite of cloud computing services, but specifically **Amazon's Elastic Compute Cloud (EC2)** and **Amazon's Relational Database Service (RDS)**. The primary difference between the two is that Amazon's EC2 is designed to be a fully-functional and fully-virtual computing environment that allows you to control as many server/database instances as you'd like (thus making it inherently scalable). Amazon's RDS, on the other hand, is designed to only act as a cloud database, though the service contains features which give you the option of scaling your computation and storage capabilities. Thus, depending on your applications, you could choose whichever service is most appropriate. Amazon's computing services are now used by many, including such high-profile startups as Yelp, Reddit, Quora, FourSquare, Hootsuite, amongst others, and is definitely something to keep in mind as you design any future backends.

The other cloud computing service is **Google's App Engine (GAE)** and is one that we'll take a more in-depth look into. Both AWS and GAE are easy to set up (relative to the traditional server method) with GAE known to be slightly more user friendly. However, the primary reason we're going to look at GAE as opposed to AWS (besides the fact that this is now a Google-themed book!) is that GAE allows you to run small-scale applications for free (up to certain predefined limits), while AWS only allows you to access their free pricing tier for a year. In this way, everyone will get to follow along as we look at more code in later sections.

Finally, the difference between the traditional server/database model and the new cloud computing model is that we don't actually need to own and manage a dedicated server! These cloud computing services allow us to essentially "rent out" server space within Amazon/Google's various data centers and allow us to quickly/cheaply create reliable, scalable applications. However, what we're giving up is some control and flexibility in the implementation, and I'll discuss this in the next section when we talk about **Google App Engine's Java Data Object (JDO)** databases.

Google App Engine and JDO databases

So what exactly is Google App Engine and why do we need it? Well, GAE is a platform that enables you to build and host web apps on the same systems that power Google applications. GAE allows us to quickly develop and deploy our applications without having to worry about reliability, scalability, hardware, patches or backups, and so on. However, this reliability and scalability comes at a cost and that cost is the flexibility with which we can select our DBMS and design our database schema. In fact, both of these are more or less chosen for you when you choose to use GAE as your backend!

GAE comes with a JDO database – meaning that it comes with a special database that allows you to directly convert Java objects into rows of data called **entities** (hence the name). This JDO database is built on top of a special web database called BigTable, which is designed to be extremely quick and scalable, and is actually not a relational DBMS like MySQL (see <http://en.wikipedia.org/wiki/BigTable>). This primarily means that not all of the features we learned in *Chapter 3, SQLite Queries*, about SQL (that is, JOINS) will be applicable here, so your design decisions regarding how your database schema should look are somewhat limited.

In light of this, Google does a nice job in providing you with a variant of SQL called **GQL**, which is a querying language designed for retrieving entities from the App Engine scalable datastore. Again, there are some differences but the general feel of GQL is much like that of SQL: where you have `SELECT` statements with `WHERE` filters and other familiar clauses like `ORDER BY` and `LIMIT`. In this way, for those who are only familiar with relational systems like MySQL, it shouldn't be terribly difficult to pick up.

For the sake of completeness, other differences include not being able to filter on multiple conditions without having to build an index, not being able to use inequality filters on multiple columns within the same query, and not being able to filter for rows with missing fields, amongst others. The reason for all of these seemingly arbitrary differences involve the architecture of the BigTable database. Because of the way it's designed and the way it indexes each row that is inserted, certain queries that are available in relational databases like MySQL will no longer be applicable with BigTable. However, because of this architecture, BigTable is inherently scalable, and so when choosing between the two just keep these trade-offs in mind.

In any case, words can only take you so far and all of these differences and similarities will become much clearer once we start seeing some actual code. Thus, in addition to having the Android SDK installed, I invite you to take some time getting Google App Engine set up using the following URL as a guide:

http://code.google.com/appengine/downloads.html#Download_the_Google_App_Engine_SDK

At this point, we're ready to dive right into some code and try to piece together a fully functional Google App Engine backend for our Android applications!

GAE: an example with video games

In the next couple of chapters, we'll be going through an example where we wish to create an application that allows us to see what video games are available through Blockbuster. This will ultimately involve everything from writing a scraper to fetch and retrieve those video games from Blockbuster's website, storing these game objects into our GAE database, writing servlets to get/remove game objects from our GAE database through HTTP requests, and last but not least finishing it off with some code for the Android client side.

In this chapter, we'll focus on setting up our database and writing wrapper methods to help us store, retrieve, update, and delete data for future steps. And so to start, every GAE application needs to first define a base entity class which essentially defines what a row is in our database. Note that each entity needs to have an ID or a key associated with it, so the only field we really need is the one for an ID. Here is the `ModelBase` class, which we will use as our base entity class and which we will override for all objects that we create:

```
@PersistenceCapable(detachable = "true")
@Inheritance(strategy = InheritanceStrategy.SUBCLASS_TABLE)
public class ModelBase {

    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Long id;

    public Long getId() {
        return id;
    }
}
```

So we'll notice that the general structure of the class resembles that of a relatively simple Java object, but that there are some odd `@` tags. Let's look at the first two:

```
@PersistenceCapable(detachable = "true")
@Inheritance(strategy = InheritanceStrategy.SUBCLASS_TABLE)
```

The first one tells us that this class needs to be `PersistenceCapable`. When you define an object as capable of being persistent, what you're telling the JDO database is that this object is capable of being stored and retrieved from the datastore. It's important to declare your entity classes as `PersistenceCapable` and then declare the desired fields as being `Persistent`. You'll see that there's also a parameter called `detachable`, which we set to be `true`. This gives us permission to edit and modify entities that we retrieved from our database even after we've closed it. Now, this does not mean that those modifications will persist in the database because it is closed, but at least we'll have permission to do so.

Next there is an `Inheritance` tag which basically means that we're allowed to create entities that override this base entity, hence inherit the base entity. The other two tags are pretty self-explanatory. The first declares that our ID (I'll quickly note that in my case I chose to use a long type as my ID but one can also use a `Key` type object) acts as the `PrimaryKey` for our entity. For people with a background in SQL this should immediately ring a bell, but basically this just tells the JDO database that objects of this entity will have a unique long ID field to be used for lookups, and so on. The last tag is one that we mentioned briefly earlier – namely the `Persistent` tag which simply tells us that this long ID field should be stored as its own column in our table.

And now, for the actual `VideoGame` object, first notice how we extend (inherit) the previous `ModelBase` class and then we continue by defining all desired persistent fields as well as implementing the constructor, and so on, as follows:

```
// NOTE HOW WE DECLARE OUR OBJECT AS PERSISTENCE CAPABLE
@PersistenceCapable
public class VideoGame extends ModelBase {

    // NOTE THE PERSISTENT TAGS
    @Persistent
    private String name;

    // USE A SPECIAL GOOGLE APP ENGINE LINK CLASS FOR URLS
    @Persistent
    private Link imgUrl;

    @Persistent
    private int consoleType;

    public VideoGame(String name, String url, String consoleType) {
        this.name = name;
        this.imgUrl = new Link(url);
        // CONVERT ALL CONSOLES TO INTEGER TYPES
        this.consoleType =
            VideoGameConsole.convertStringToInt(consoleType);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Link getImgUrl() {
        return imgUrl;
    }
}
```

```
public void setImgUrl(Link imgUrl) {
    this.imgUrl = imgUrl;
}

public int getConsoleType() {
    return consoleType;
}

public void setConsoleType(int consoleType) {
    this.consoleType = consoleType;
}

public static class VideoGameConsole {

    public static final String XBOX = "Xbox";
    public static final String PS3 = "Ps3";
    public static final String WII = "Wii";
    public static final String PSP = "Psp";
    public static final String DS = "NintendoDS";
    public static final String PS2 = "Ps2";

    public static final String[] CATEGORIES = { "Xbox", "Ps3",
        "Wii", "Psp", "NintendoDS", "Ps2" };

    public static int convertStringToInt(String type) {
        if (type == null) { return -1; }
        if (type.equalsIgnoreCase(XBOX)) {
            return 0;
        } else if (type.equalsIgnoreCase(PS3)) {
            return 1;
        } else if (type.equalsIgnoreCase(PS2)) {
            return 2;
        } else if (type.equalsIgnoreCase(PSP)) {
            return 3;
        } else if (type.equals(WII)) {
            return 4;
        } else if (type.equals(DS)) {
            return 5;
        } else {
            return -1;
        }
    }
}
}
```

Once you get the gist of what the @ tags are doing, the rest is pretty self-explanatory. Here I'm simply declaring a few fields as persistent, and then I implement a constructor as well as a convenient inner class. The reason why I like having a convenience class (that is, `VideoGameConsole` in this case) is that typically in tables, querying for integers is a lot more efficient and reliable than querying for strings (one: you don't need to worry about case matching, and two: integer comparisons are just much more efficient in general than string comparisons). And so, ideally I'd like a way to convert strings to integers and potentially even be able to map a group of strings to an integer (that is, "PS3" could get mapped to 1, and so could "Playstation 3" or "PS 3").

Now that we have our `VideoGame` entity defined, we're ready to start implementing our database and telling it how to interact with these `VideoGame` entities.

The PersistenceManager and Queries

So the first step is defining a way to establish a connection between the server and the database. Remember back at the beginning of the book when we had to call methods such as `getWritableDatabase()` before making any queries? Well the same is true here, but instead of using a `SQLiteOpenHelper` class, we will define a `PersistenceManager` class as follows:

```
public final class PMF {

    private static final PersistenceManagerFactory pmfInstance =
        JDOHelper.getPersistenceManagerFactory("transactions-optional");

    private PMF() {
    }

    public static PersistenceManagerFactory get() {
        return pmfInstance;
    }

}
```

Notice that it is defined as a singleton for improved efficiency, and all we're doing is opening a persistence (database) manager that can handle transactions (queries). Then in our future queries, we no longer need to sacrifice performance by repeatedly requesting for a `PersistenceManager` and instead can grab the existing instance.

Once we have our `PersistenceManager` defined, we can start implementing our series of wrappers and we'll begin by looking at how to insert new game objects:

```
public class VideoGameJDOWrapper {

    /**
     * INSERT A SINGLE VIDEOGAME OBJECT
     *
     * @param g
     *         - a video game object
     */
    public static void insertGame(VideoGame g) {
        PersistenceManager pm = PMF.get().getPersistenceManager();
        try {
            pm.makePersistent(g);
        } finally {
            pm.close();
        }
    }

    /**
     * INSERT MULTIPLE VIDEOGAME OBJECTS - MORE EFFICIENT METHOD
     *
     * @param games
     *         - a list of video game objects
     */
    public static void batchInsertGames(List<VideoGame> games) {
        PersistenceManager pm = PMF.get().getPersistenceManager();
        try {
            // ONLY NEED TO RETRIEVE AND USE PERSISTENCEMANAGER ONCE
            pm.makePersistentAll(games);
        } finally {
            pm.close();
        }
    }
}
```

Not too bad right? The idea is simple and is one that we have seen earlier – simply grab our instance of the `PersistenceManager` (that is, our connection to the database) and make the `VideoGame` object that's passed in persistent. Again, remember that when working with GAE, the idea of persistence is the same as insertion and so by making an object persistent, we are literally telling the database to convert our entity into a row of our `VideoGame` table. We can also see that when adding many entities at once, GAE offers us an efficient way to do so by using batch inserts. Now let's take a look at how we would get video game objects from our database. Querying for entities is much more involved than simply inserting entities, but instead of devoting an entire chapter to all the different ways you can submit queries (like we did in *Chapter 3, SQLite Queries*) I'll just show you one convenient and intuitive way to do it, and if you're curious I invite you to check out:

<http://code.google.com/appengine/docs/java/datastore/queries.html>

But yes, here's one way to do it and it should remind you of our previous encounters with the `SQLiteQueryBuilder` class:

```
public class VideoGameJDOWrapper {

    public static void insertGame(VideoGame g) {
        . . .
    }

    public static void batchInsertGames(List<VideoGame> games) {
        . . .
    }

    /**
     * GET ALL VIDEO GAMES OF A CERTAIN PLATFORM
     *
     * @param platform
     *        - desired platform of games
     * @return
     */
    public static List<VideoGame> getGamesByType(String platform) {
        PersistenceManager pm = PMF.get().getPersistenceManager();

        // CONVERT STRING OF PLATFORM TO INT TYPE
        int type = VideoGameConsole.convertStringToInt(platform);

        // INIT A NEW QUERY AND SPECIFY THE OBJECT TYPE
        Query query = pm.newQuery(VideoGame.class);
```

```

        // SET THE FILTER - EQUIVALENT TO SQL WHERE FILTER
        query.setFilter("consoleType == inputType");

        // TELL THE QUERY WHAT PARAMETERS YOU WILL SEND
        query.declareParameters("int inputType");
        List<VideoGame> ret = null;
        try {
            // EXECUTE QUERY WITH PARAMETERS
            ret = (List<VideoGame>) query.execute(type);
        } finally {
            // CLOSE THE QUERY AT THE END
            query.closeAll();
        }
        return ret;
    }

    /**
     * GET ALL VIDEO GAMES OF A GIVEN PLATFORM WITH A LIMIT ON THE
     * NUMBER OF
     * RESULTS
     *
     * @param platform
     *         - desired platform of games
     * @param limit
     *         - max number of results to return
     * @return
     */
    public static List<VideoGame> getGamesByTypeWithLimit
    (String platform, int limit) {
        int type = VideoGameConsole.convertStringToInt(platform);
        PersistenceManager pm = PMF.get().getPersistenceManager();
        Query query = pm.newQuery(VideoGame.class);
        query.setFilter("consoleType == inputType");
        query.declareParameters("int inputType");

        // SAME QUERY AS ABOVE BUT THIS TIME SET A MAX RETURN LIMIT
        query.setRange(0, limit);
        List<VideoGame> ret = null;
        try {
            ret = (List<VideoGame>) query.execute(type);
        } finally {
            query.closeAll();
        }
        return ret;
    }
}

```



```
/**
 * QUICKEST WAY TO RETRIEVE OBJECT IF YOU HAVE THE ID
 *
 * @param id
 *         - row id of the object
 * @return
 */
public static VideoGame getVideoGamesById(long id) {
    PersistenceManager pm = PMF.get().getPersistenceManager();
    return (VideoGame) pm.getObjectById(VideoGame.class, id);
}

}
```

Let's dissect the first method piece by piece:

```
PersistenceManager pm = PMF.get().getPersistenceManager();

// CONVERT STRING OF PLATFORM TO INT TYPE
int type = VideoGameConsole.convertStringToInt(platform);

// INIT A NEW QUERY AND SPECIFY THE OBJECT TYPE
Query query = pm.newQuery(VideoGame.class);
```

Here, we grab our `PersistenceManager` and then we convert the passed-in platform into an integer type, since we're going to filter by platform. Next, we tell our `PersistenceManager` that we want to open a new query (that is, start a new `SELECT` statement) and so we call our `newQuery()` method. Then, we set the details of our query with the following methods:

```
// SET THE FILTER - EQUIVALENT TO SQL WHERE FILTER
query.setFilter("consoleType == inputType");

// TELL THE QUERY WHAT PARAMETERS YOU WILL SEND
query.declareParameters("int inputType");
```

Here we first set our filter and specify on which column we want to perform the filtering (that is, setting the `WHERE` part of our query). Next, we set a placeholder for the parameters that will get passed in (think of the `?` placeholders from earlier) and lastly, we execute the query and pass in the platform type parameter. In the method that follows, everything remains the same except for an additional `LIMIT` filter, set using the following method:

```
query.setRange(0, limit);
```

The third method we implemented is relatively straightforward – the JDO database allows you to quickly retrieve an entity if you have their unique key or ID by calling the `PersistenceManager`'s `getObjectById()` method. Again, there are many ways to execute queries in GAE as well as many other clauses and subtleties that I won't go into in this book, but for now you should have the basic idea down and should be able to execute the vast majority of queries needed. Finally, let's take a look at how we would update and delete objects from our database:

```
public class VideoGameJDOWrapper {

    public static void insertGame(VideoGame g) {

    }

    public static void batchInsertGames(List<VideoGame> games) {

    }

    public static List<VideoGame> getGamesByType(String platform) {

    }

    public static List<VideoGame> getGamesByTypeWithLimit
    (String platform, int limit) {
        . . .
    }

    public static VideoGame getVideoGamesById(long id) {
        . . .
    }

    /**
     * METHOD FOR UPDATING THE NAME OF A VIDEO GAME
     *
     * @param newName
     *         - new name of the game
     * @param id
     *         - the row id of the object
     * @return
     */
    public static boolean updateVideoGameName(String newName, long id)
    {
```

```
PersistenceManager pm = PMF.get().getPersistenceManager();
boolean success = false;
try {
    // AS LONG AS PERSISTENCE MANAGER IS OPEN THEN ANY CHANGES
    // TO OBJECT
    // WILL AUTOMATICALLY GET UPDATED AND STORED
    VideoGame v = (VideoGame) pm.getObjectById(VideoGame.
class, id);
    if (v != null) {
        // KEEP PERSISTENCEMANAGER OPEN
        v.setName(newName);
        success = true;
    }
} catch (JDOObjectNotFoundException e) {
    e.printStackTrace();
    success = false;
} finally {
    // ONCE CHANGES ARE MADE - CLOSE MANAGER
    pm.close();
}
return success;
}

/**
 * DELETE ALL GAMES OF A CERTAIN PLATFORM
 *
 * @param platform
 *      - specify the platform of the games you want to delete
 */
public static void deleteGamesByType(String platform) {
    PersistenceManager pm = PMF.get().getPersistenceManager();
    int type = VideoGameConsole.convertStringToInt(platform);

    // INIT QUERY AGAIN
    Query query = pm.newQuery(VideoGame.class);

    // SAME WHERE FILTERS
    query.setFilter("consoleType == inputType");
    query.declareParameters("int inputType");

    // NOW CALL THE DELETE METHOD
    query.deletePersistentAll(type);
}
}
```

Again, let's take the first method – our update method – and dissect it piece by piece:

```
PersistenceManager pm = PMF.get().getPersistenceManager();
boolean success = false;
try {
    VideoGame v = (VideoGame) pm.getObjectById(VideoGame.class, id);
    if (v != null) {
        // KEEP PERSISTENCEMANAGER OPEN
        v.setName(newName);
        success = true;
    }
}
```

So just like in the previous example, we start by getting our connection with the JDO database. Then we try to retrieve our `VideoGame` object by calling the `getObjectById()` method and passing in the unique ID of the entity we want to update. Here's one of the odd things about the `PersistenceManager` that you should keep in mind.

Instead of having an explicit update method, which we are used to seeing by now, with a `PersistenceManager` as long as the connection is open any changes you make to the object will automatically get updated in the database. So notice that in this method, the first step is to retrieve the entity, update it while the connection is still open, and then close the connection once the entity has been updated.

Of course in this example we only update a specific ID at a time, but one can see how by keeping this detail in mind, we can easily write a method that updates a group of entities at once – simply query for a list of them and update each one while the `PersistenceManager` is still open.

And last but not least, for our delete method, we see that all the steps are the same as the previous get methods, except for the last line, where we use the method:

```
// NOW CALL THE DELETE METHOD
query.deletePersistentAll(type);
```

Otherwise, all of the prior logic stays the same. And that's it! Now we have a JDO database wrapper class that allows us to abstract away all of the messy `PersistenceManager` syntax, and which gives us a quick way to insert, retrieve, update, and delete data from our GAE backend! The next step then is to actually figure out a way to retrieve this video game data, at which point we can simply wrap it in our `VideoGame` entity class and push it into our database.

Summary

In this chapter, we moved away from the Android platform and started expanding upon our understanding of external databases. We began by taking a cursory look at what our options were: the traditional dedicated server with the database connection (for instance an Apache Tomcat server hooked up to a MySQL server) or a cloud computing server/database combination, such as **Amazon Web Services (AWS)** or **Google App Engine (GAE)**.

Google App Engine is nice in that it's much easier to set up and also allows us to build simple, relatively small-scale applications free of cost and time constraints. Both cloud-computing solutions come with reliable servers as well as efficient, scalable databases, but limit the amount of control you have over your backend – especially when compared to the unlimited freedom you have when you buy your own dedicated server.

Sticking with GAE, we started building out a simple video games application that shows us all the games available through Blockbuster. We introduced the notion of persistence in GAE and wrote our first entity class. We then wrote our own `PersistenceManager` singleton class and implemented a convenience class for getting, inserting, updating, and deleting data from our database.

We covered a lot of ground in this chapter, but we still have a long way to go before having a complete, fully functional application. In the next chapter, we'll look at ways to retrieve data and then store it using the wrapper methods written in this chapter.

9

Collecting and Storing Data

Onwards we go! In the previous chapter, we introduced a couple of popular external databases that you could use and decided to develop a fully functional backend using Google's App Engine (GAE). We managed to create a new project on GAE and use the `PersistenceManager` to build out an extremely useful wrapper class that illustrated some of the concepts central to our JDO database. This wrapper class will soon be extremely handy to have around as we start inserting real data and subsequently query that data using our Android application.

So here we are—the next step! For most people trying to build out a data-centric application, actually getting that data will be extremely difficult and will typically require a lot of time and money. However, there are many tools and methods at our disposal which can help us use existing data to fill up our databases. In this next chapter, we'll take a look at some of those methods, and will finish by inserting our newly acquired data into our JDO database.

Methods for collecting data

To begin, let's briefly go over two different ways in which you can collect data:

- Through an Application Programming Interface (API)
- Through web scraping

The first and simplest way is through using an API. For those who have never used an API before, think of this as a *web library* created by some third-party company, which typically allows you to call a handful of functions (almost always executed as HTTP requests), which then give you access to a subset of their data.

For instance, a common API is the Facebook Graph API which, when authenticated, allows you to query for a user's profile information or an event's details, and so on. Essentially, through the API, I can access the same data about a person or event that I would see on Facebook's website, just through a different channel. This is what I mean by the company *exposing* a subset of their data. Another example might be with Yelp, whose API allows you to query for restaurants and venues when passed a set of parameters (that is, location). Here, even though I'm not actually on Yelp's web page, I can still access their data through their API.

Having an API available to collect your data is extremely useful because of how the data is already there and ready for you to use; depending on the credibility of the company, oftentimes the data will already be cleaned and well formatted. This saves you from having to find the data on your own and subsequently clean the data on your own. The catch, however, is that oftentimes companies will not allow you to store their data for proprietary reasons, and so depending on what your application does, you may need to keep this legal issue in mind.

So what exactly happens when no API is available for you to use? Well, then you'll have to resort to getting that data on your own, and one great way to do that is through **web scraping**. In the next section, I'll devote a great deal of time to explaining what the art of web scraping is and how you go about doing it. For now, let's end this short section with a discussion on the two popular formats in which data is often returned by APIs.

The first is called Extensible Markup Language (XML) and is a human-readable and machine-readable data format that takes the form of a tree and looks very similar to HTML actually. A simple example of what this tree structure looks like is say you call the Facebook Graph API and it returns a list of your friends. The root of the tree might have the tag `<friends>`, and underneath it may have a series of leaves with the tag `<friend>`. Then, each `<friend>` node might branch off into several descriptor tags such as `<name>`, `<age>`, and so on. In fact, in the examples later on, I'll actually use XML as the data format of choice because of how it's human readable, so you'll get to see real examples of what this looks like.

The next is called **JavaScript Object Notation (JSON)** and it is a much more lightweight data structure than XML. JSON is still machine readable but is less friendly for human readability. The trade-off though is that parsing JSON tends to be more efficient, and so really the decision between which to use just depends on how important human readability is relative to performance. The general structure of JSON resembles that of a map instead of a tree. Using the same preceding example, instead of being returned a tree structure with `<friends>` as the root node, we might have `friends` as a key with value equal to a JSON array. The JSON array would then have a list of `friend` keys, each of which has a value equal to a JSON object. Finally, the JSON object would have keys equal to `name`, `age`, and so on. In other words, you can think of JSON structures as series of embedded maps, where many times keys will point to a sub-map, which then has its own keys, and so on.

So often when using third-party APIs, you'll need to be aware of which data format they choose to return their data in, and parse the results accordingly. Furthermore, even when you're implementing web scrapers and finding yourself having to build your own API, it often helps to pick one of the two data formats and stick with it. This will make your life a lot simpler when it comes to calling your own API from external applications and then parsing the returned result. Now, moving on to web scraping.

A primer on web scraping

Web scraping is the art of structuring web HTML and methodically parsing data from it. The idea is that HTML should be (to some extent) inherently well structured, as every open tag (that is, ``) should be followed by a close tag (that is, ``). In this way, HTML if structured correctly, can be viewed as a tree structure very much like XML often is. Scraping a website can be achieved in any number of ways, which typically vary with the complexity of the underlying HTML source code, but at a high level, it involves three steps:

1. Obtain the desired URL, establish a connection to the URL, and retrieve its source code.
2. Structure and clean the underlying source code so that it becomes a valid XML document.
3. Run a tree-navigating language like XPath (or XQuery and XSLT), and/or use regular expressions (REGEX) to parse out desired nodes.

The first step is relatively self-explanatory, but I will note one thing. Often you'll find yourself needing to scrape some sort of dynamic web page, meaning that the URL is not going to be static and may change depending on the date, some set of criteria, and so on. Let's walk through two examples of what I mean here.

The first involves stocks. Let's say you're trying to write a web scraper that can scrape for the current price of a given stock, say from Yahoo! Finance. Well, first off, what does the URL look like? Checking really quickly for Google's (ticker GOOG) current price, we see that the URL of the corresponding web page is:

```
http://finance.yahoo.com/q?s=GOOG
```

It's a pretty simple URL and we'll quickly notice that the ticker of the stock gets passed as a parameter to the URL. In this case, the parameter has key `s` and value equal to the ticker. Now it's pretty easy to see how we can quickly construct a dynamic URL to solve our problem—all we would have to do is write a simple method as follows:

```
public void stockScraper(String ticker) {  
    String URL_BASE = "http://finance.yahoo.com/q?s=";  
    String STOCK_URL = URL_BASE + ticker;  
  
    // CONTINUE SCRAPING STOCK_URL  
}
```

Neat, right? Now let's say we don't just want the current stock price, but we want to pull all historical prices between two dates. Well, first let's take a look at what a sample URL would be, again for Google's stock:

```
http://finance.yahoo.com/q/hp?s=GOOG&a=07&b=19&c=2004&d=02&e=14  
&f=2012
```

So what do we notice here? We notice that the ticker is still being passed as a parameter with key `s`, but in addition to that we notice what looks like two distinct dates being passed with various keys. The dates look like 07/19/2004, most likely the start date, and 02/14/2012, what appears to be the end date, and they seem to have key values `a` through `f`. In this case, the key values aren't the most intuitive, and oftentimes you'll see key values of `day` or `d` and `month` or `m` instead. However, the idea is simply that with this URL, not only can you dynamically adjust what the ticker is but depending on what range of dates your user is looking for, you can adjust those as well. By keeping this idea in mind, you'll slowly learn how to better decipher various URLs and learn how to make them extremely dynamic and suitable for your scraping needs.

Now, some websites make their requests through POST requests. The difference is that in POST requests, the parameters are embedded within the request (as opposed to being embedded within the URL). This way, potentially private data is not visibly displayed in the URL (though this is just one use case for POST requests). So what do we do when this is the case? Well, there's no terribly easy answer. Typically, you'll need to download an HTTP request listener (for browsers like Chrome and Firefox, simply search for an HTTP request listener add-on). This will then allow you to see what requests are being made (both GET and POST requests), as well as the parameters that were passed. Once you know what the parameters are, then the rest works just like a GET request.

Now, once we have our URL, the next step is to get the underlying source code and structure it. Of course, this can be a pain to do yourself, but fortunately, there are libraries out there which will clean and structure the source code for us. The one that I most frequently use is called **HtmlCleaner** and can be found at the following URL:

<http://htmlcleaner.sourceforge.net/>

It's a great library that gives you methods for cleaning and structuring the source code, navigating the resulting XML document, and ultimately parsing the values and attributes of the XML nodes. Once our data is cleaned, the last step is simply to walk through the tree and pick out the pieces of data we want. Now, this is easier said than done, as there's no really easy way to traverse the tree methodically and reliably using just Java and its native packages. What I mean by methodically and reliably is being able to traverse the tree and parse the correct data even when the structure of the underlying source code has changed slightly.

For instance, say your parsing method was as naive as telling your code to give you the value of the fifth node. What happens then, when Yahoo! (or whatever site you're scraping) decides to add a new header to their website, and now the fifth node becomes the sixth? Even under this relatively simple change to the underlying, your scraper will break and will start returning you values from an incorrect node, and so ideally, we'd like to find a method for getting the correct node value regardless of how the underlying website changes.

Luckily for us, oftentimes frontend engineers will build websites where important fields will have tags that contain either `class` or `id` attributes with unique values. We can then take advantage of these helpful and descriptive naming conventions and use a nifty language called **XPath**. The language itself is fairly self-explanatory once you see it; in fact, the syntax resembles that of any path (that is, directory path, URL path, and so on), so I'll simply direct you to the following URL to learn the ins and outs, if you wish:

<http://www.w3schools.com/xpath/>

In any case, for now just keep in mind that XPath is a simple language that allows you to return sets of nodes which are determined by a path. What's special about XPath is that within the path, you can further refine your search by including various filters, ones that allow us to return only those `div` that are of a certain `class` for instance. This is where having descriptive `class` and `id` attributes comes in handy because we can drill into the HTML and efficiently find only those nodes that are important to us. Furthermore, if you still need additional weapons to parse the resulting XML, you could include regular expressions (REGEX) to help you in your search.

In the end, the idea is to be as robust as possible with your parsing, as the last thing you want to do is to have to update your scrapers constantly as small, insignificant changes are made to the underlying website. Again, sometimes the website changes dramatically and you'll legitimately have to update your scraper, but the idea is to write them, again, as robustly as possible.

At this point I'm sure you have plenty of questions. What does the code actually look like? How do you grab a website's HTML? How do you even use the `HtmlCleaner` library? What's an example of XPath? Previously, my goal was to lead you to a high-level understanding of what web scraping is, and along the way, I introduced a lot of different technologies and techniques that one would use. Now, let's get our hands dirty with some code and see each of the preceding steps in action. Here are steps one and two for scraping our Blockbuster video games data:

```
public class HTMLNavigator {

    // STEP 1 - GET THE URL'S SOURCE CODE
    public static CharSequence navigateAndGetContents(String url_str)
    throws IOException {
        URL url = new URL(url_str);

        // ESTABLISH CONNECTION TO URL
        URLConnection conn = url.openConnection();
        conn.setConnectTimeout(30000);
        String encoding = conn.getContentEncoding();
        if (encoding == null) {
            encoding = "ISO-8859-1";
        }

        // WRAP BUFFERED READER AROUND INPUT STREAM
        BufferedReader br = new BufferedReader
            (new InputStreamReader(conn.getInputStream(), encoding));
        StringBuilder sb = new StringBuilder();
        try {
            String line;
```

```

        while ((line = br.readLine()) != null) {
            sb.append(line);
            sb.append('\n');
        }
    } finally {
        br.close();
    }
    return sb;
}
}

```

So first we have a simple convenience class that allows us to get the source code of a passed-in URL. It simply opens a connection, sets a few standard web parameters, and then reads the input stream. We use a `StringBuilder` to efficiently construct one large string containing each line of the input stream, and finally close all connections and return the string. This string will then be the underlying HTML of the passed-in URL, and is what we'll need in the next step to construct a clean, organized XML document. The code for that is as follows:

```

import org.htmlcleaner.CleanerProperties;
import org.htmlcleaner.HtmlCleaner;
import org.htmlcleaner.TagNode;
import org.htmlcleaner.XPatherException;

import app.helpers.HTMLNavigator;
import app.types.VideoGame;

public class VideoGameScraper {

    private static String content;

    private static final String BASE_URL = "http://www.blockbuster.com/
games/platforms/gamePlatform";

    /**
     * QUERY FOR GAMES OF CERTAIN PLATFORM
     *
     * @param type
     *           the platform type
     * @return
     * @throws IOException
     * @throws XPatherException
     */
}

```

```
public static List<VideoGame> getVideoGamesByConsole(String type)
throws IOException, XPatherException {
    // CONSTRUCT FULL URL
    String query = BASE_URL + type;

    // STEPS 1 + 2 - GET AND CLEAN THE DYNAMIC URL
    TagNode node = getAndCleanHTML(query);

    // STEP 3 - PARSE AND ADD GAMES
    List<VideoGame> games = new ArrayList<VideoGame>();

    . . .

    return games;
}

/**
 * CLEAN AND STRUCTURE THE PASSED IN HTML
 *
 * @param result
 *         the underlying html
 * @return
 * @throws IOException
 */
private static TagNode getAndCleanHTML(String result) throws
IOException {
    String content = HTMLNavigator.navigateAndGetContents(result).
toString();
    VideoGameScraper.content = content;

    // USE HTMLCLEANER TO STRUCTURE HTML
    HtmlCleaner cleaner = new HtmlCleaner();
    CleanerProperties props = cleaner.getProperties();
    props.setOmitDoctypeDeclaration(true);
    return cleaner.clean(content);
}

.
.
.
}
```

And so here we first write a simple method which allows us to connect to the resulting URL and grab its underlying source code. We then take that result and pass it to a cleaning method which instantiates a new instance of our `HtmlCleaner` class and calls the `clean()` method. This method will structure the underlying HTML into a well-formed XML document, and return the root of the XML as a `TagNode` object. The last step is simply looking at the underlying source code, determining what the correct XPaths are, and then running those over the given root `TagNode`. The abridged source code of Blockbuster's video game rental page looks like the following code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html lang="en" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
<body class="full">
<script type="text/javascript">
<div class="body clearDiv">
<div id="pageMask">&nbsp;</div>
<div id="boxPopup">&nbsp;</div>
<div id="head" class="head">
<style type="text/css">
<div>
<div id="gamesNav" class="secondaryNav">
<script type="text/javascript" language="javascript">

<div class="page clearDiv">
<div class="main contentsMain clearDiv">
<div class="primary clearDiv">
<span class="contentsDM"></span>
<span class="contentsLB"></span>

<h1>Action & Adventure Video Games</h1>
<div class="pagination">
<div class="gb6 listViewHeader">
<div class="">

<div id="4453" class="addToQueueEligible game sizeb gb6 bvr-
gamelistitem ">
<mkt marketingitemid="4453" catalystinfo="A" listname="gameActionAdve
nture"></mkt>
<a onmouseover="if(DndUtil.windowLoaded){ new GameRollover(this); }"
href="/games/catalog/gameDetails/4136" title="Superman Returns: The
Video Game">
```

```

</a>

<div class="details">
<h4>
<a onmouseover="if (DndUtil.windowLoaded){new GameRollover(this);}"
href="/games/catalog/gameDetails/4136" title="Superman Returns: The
Video Game">Superman Returns: The Video Game</a>
</h4>
<dl class="release">
<dl class="rated">
<div class="platform">
<dl class="movieInfo">
<div class="summary ">
<p class="readMore">
<div class="rolloverDetailsDiv" contentsrc="/esi/catalog/
gameRollover/4136/false">&nbsp;</div>
</div>
<div class="movieOptions">
<div id="movieRating" class="ratingWidget">
</div>
</div>
...
```

However, note that that this source code is as of the date I'm writing this and is not guaranteed to remain the same. However, from this source code above, we can see that each game is listed in a div tag with class `addToQueueEligible game sized gb6 brv-gamelistitem`. This is somewhat of a long class name but we can have some confidence that by searching for divs with this class tag, we'll find video games and only video games because of how the class tag involves adding eligible games to a queue.

Now, once we get to those desired divs, we see that the nodes we want are simply the first a node, as well as that a node's corresponding `img` tag. Hence, in order to get the title and image URLs, respectively, our desired XPath should look as follows:

```
//div[@class='addToQueueEligible game sized gb6 brv-
gamelistitem']/a[1]
//div[@class='addToQueueEligible game sized gb6 brv-
gamelistitem']/a[1]/img
```

With this, let's now take a look at the full code of our scraper:

```
import org.htmlcleaner.CleanerProperties;
import org.htmlcleaner.HtmlCleaner;
import org.htmlcleaner.TagNode;
import org.htmlcleaner.XPatherException;

import app.helpers.HTMLNavigator;
import app.types.VideoGame;

public class VideoGameScraper {

    private static String content;

    // XPATH FOR GETTING TITLE NAMES
    private static String TITLE_EXPR = "//div[@class='%s']/a[1]";

    // XPATH FOR GETTING IMAGE URLS
    private static String IMG_EXPR = "//div[@class='%s']/a[1]/img";

    // BASE OF BLOCKBUSTER URL
    public static final String BASE_URL = "http://www.blockbuster.com/
    games/platforms/gamePlatform";

    /**
     * QUERY FOR GAMES OF CERTAIN PLATFORM
     *
     * @param type
     *         the platform type
     * @return
     * @throws IOException
     * @throws XPatherException
     */
    public static List<VideoGame> getVideoGamesByConsole(String type)
    throws IOException, XPatherException {
        // CONSTRUCT FULL URL
        String query = BASE_URL + type;

        // USE HTMLCLEANER TO STRUCTURE HTML
        TagNode node = getAndCleanHTML(query);

        // ADD GAMES
        List<VideoGame> games = new ArrayList<VideoGame>();
```



```
        games.addAll(grabGamesWithTag(node, "addToQueueEligible
game sizeb gb6 bvr-gamelistitem    ", type));
        return games;
    }

    /**
     * GIVEN THE STRUCTURED HTML, PARSE OUT NODES OF THE PASSED IN TAG
     *
     * @param head
     *         the head of the structured html
     * @param tag
     *         the tag we are looking for
     * @param type
     *         the platform type
     * @return
     * @throws XPatherException
     */
    private static List<VideoGame> grabGamesWithTag(TagNode head,
String tag, String type) throws XPatherException {
        // RUN VIDEO GAME TITLE AND IMAGE XPATHS
        Object[] gameTitleNodes = head.evaluateXPath(String.format
(TITLE_EXPR, tag));
        Object[] imgUrlNodes = head.evaluateXPath(String.format
(IMG_EXPR, tag));

        // ITERATE THROUGH VIDEO GAMES
        List<VideoGame> games = new ArrayList<VideoGame>();
        for (int i = 0; i < gameTitleNodes.length; i++) {
            TagNode gameTitleNode = (TagNode) gameTitleNodes[i];
            TagNode imgUrlNode = (TagNode) imgUrlNodes[i];
            // BY LOOKING AT THE HTML, WE CAN DETERMINE
            // WHICH ATTRIBUTES OF THE NODE TO PULL
            String title = gameTitleNode.getAttributeByName("title");
            String imgUrl = imgUrlNode.getAttributeByName("src");

            // BUILD OUR VIDEO GAME OBJECT AND ADD TO LIST
            VideoGame v = new VideoGame(title, imgUrl, type);
            games.add(v);
        }
        return games;
    }
}
```

```

/**
 * CLEAN AND STRUCTURE THE PASSED IN HTML
 *
 * @param result
 *         the underlying html
 * @return
 * @throws IOException
 */
private static TagNode getAndCleanHTML(String result) throws
IOException {
    . . .
}

```

And that's it! Most of this code we've already seen earlier, so really it's just the `grabGamesWithTag()` method that we should hone in on. The first part of the method is to take the HTML patterns that we saw earlier (in the source code of the website) and combine them with our XPath formats. At this point, we have a valid XPath that will lead us to both the titles of the video games, as well as to the image URLs of the video games. The method from `HtmlCleaner` that we need to use to actually run this XPath command is as follows:

```

Object[] gameTitleNodes = head.evaluateXPath(String.format
(TITLE_EXPR, tag));

```

This will return a list of `Objects` which can then be cast to individual `TagNode` objects. What we need to do then is loop through each `Object` in our array, cast it to a `TagNode`, and extract either the value of the node or an attribute of the node to obtain the desired data. We can see that in the following part of the method:

```

// ITERATE THROUGH VIDEO GAMES
List<VideoGame> games = new ArrayList<VideoGame>();
for (int i = 0; i < gameTitleNodes.length; i++) {
    TagNode gameTitleNode = (TagNode) gameTitleNodes[i];
    TagNode imgUrlNode = (TagNode) imgUrlNodes[i];
    // BY LOOKING AT THE HTML, WE CAN DETERMINE
    // WHICH ATTRIBUTES OF THE NODE TO PULL
    String title = gameTitleNode.getAttributeByName("title");
    String imgUrl = imgUrlNode.getAttributeByName("src");

    // BUILD OUR VIDEO GAME OBJECT AND ADD TO LIST
    VideoGame v = new VideoGame(title, imgUrl, type);
    games.add(v);
}

```

In both cases here, the values that we need are specific attributes of the node, as opposed to the value of the node. Had it been a value, our code would have looked more like the following:

```
List<VideoGame> games = new ArrayList<VideoGame>();
for (int i = 0; i < gameTitleNodes.length; i++) {
    TagNode gameTitleNode = (TagNode) gameTitleNodes[i];
    TagNode imgUrlNode = (TagNode) imgUrlNodes[i];

    String title = gameTitleNode.getText().toString();
    String imgUrl = imgUrlNode.getAttributeByName("src");

    // BUILD OUR VIDEO GAME OBJECT AND ADD TO LIST
    VideoGame v = new VideoGame(title, imgUrl, type);
    games.add(v);
}
```

At this point, we've run through a quick primer on web scraping. Again, web scraping is a technique and an art that will take time to get used to and master, but is a great skill to have and is one that will open up countless opportunities for mining data across the Web. For now, focus on the concepts that were introduced in this chapter, as opposed to the actual code. The reason I say this is because how your code looks will very much depend on what web page you're trying to scrape. What won't change are the concepts behind the scraping, and so use those three steps mentioned in this chapter as a guide to how you can write a scraper for any web page.

Extending HTTP servlets for GET/POST methods

Now that we have our web scraper written, we need a way to take the `VideoGame` objects that are returned, and actually store them in our database. Furthermore, we need a way to communicate with our server once it's up and running and tell it to scrape the site and insert it into our JDO database. Our gateway for communicating with our server is through what's called an HTTP servlet – something that we briefly mentioned earlier in the book.

Setting up your backend in this way will be especially useful when we talk later about CRON jobs which, in order to automatically run some kind of function, require a servlet to communicate with (more on this soon). For now though, let's see how we can extend the `HttpServlet` class and implement its `doGet()` method, which will listen and handle all HTTP GET requests sent to it. But first, what exactly is an HTTP GET request? Well, an HTTP web request is simply a user making a request to some server that will be sent over the network (that is, the Internet). Depending on the type of request, the server will then handle and send an HTTP response back to the user. There are then two common types of HTTP requests:

- GET request – web requests that are only meant to retrieve data. These web requests will typically ask the server to query for some kind of data to be returned.
- POST request – web requests that submit data to be processed. Typically, this will ask the server to insert some kind of data that was submitted by the user.

In this case, since we aren't getting any data for a user or submitting any data from a user (in fact we're not really interacting with any users at all), it really doesn't make a difference which type of request we use, so we'll stick with the simpler GET request as follows:

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// EXTEND THE HTTPSERVLET CLASS TO MAKE THIS METHOD AVAILABLE
// TO EXTERNAL WEB REQUESTS, NAMELY CLIENTS AND CRON JOBS
public class VideoGameScrapeServlet extends HttpServlet {

    private ArrayList<VideoGame> games;

    /**
     * METHOD THAT IS HIT WHEN HTTP GET REQUEST IS MADE
     *
     * @param request
     * a servlet request object (any params passed can be retrieved
     * with this)
     * @param response
     * a servlet response that you can embed data back to user
     */
}
```

```
* @throws IOException
* @throws ServletException
*/
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {
    games = new ArrayList<VideoGame>();
    String message = "Success";
    try {
        // GRAB GAMES FROM ALL PLATFORMS
        games.addAll(
            VideoGameScraper.getVideoGamesByConsole(VideoGameConsole.DS));
        games.addAll(
            VideoGameScraper.getVideoGamesByConsole(VideoGameConsole.PS2));
        games.addAll(
            VideoGameScraper.getVideoGamesByConsole(VideoGameConsole.PS3));
        games.addAll(
            VideoGameScraper.getVideoGamesByConsole(VideoGameConsole.PSP));
        games.addAll(
            VideoGameScraper.getVideoGamesByConsole(VideoGameConsole.WII));
        games.addAll(
            VideoGameScraper.getVideoGamesByConsole(VideoGameConsole.XBOX));
    } catch (Exception e) {
        e.printStackTrace();
        message = "Failed";
    }

    // HERE WE ADD ALL GAMES TO OUR VIDEOGAME JDO WRAPPER
    VideoGameJDOWrapper.batchInsertGames(games);

    // WRITE A RESPONSE BACK TO ORIGINAL HTTP REQUESTER
    response.setContentType("text/html");
    response.setHeader("Cache-Control", "no-cache");
    response.getWriter().write(message);
}
}
```

So the method itself is quite simple. We already have our `getVideoGamesByConsole()` method from earlier, which goes and does all the scraping, returning a list of `VideoGame` objects as a result. We then simply run it for every console that we want, and at the end use our nifty JDO database wrapper class and call its `batchInsertGames()` method for quicker insertions. Once that's done, we take the HTTP response object that is passed in and quickly write some kind of message back to the user to let them know whether or not the scraping was successful. In this case, we don't make use of the `HttpServletRequest` object that gets passed in, but that object will come in very handy if the requester passes parameters into the URL. For instance, say you wanted to write your servlet in a way that only scrapes one specific game platform instead of all of them. In that case, you would need some way of passing a platform-type parameter to your servlet, and then extracting that passed-in parameter value within the servlet. Well, just like how earlier we saw that Yahoo! Finance allows you to pass in tickers with key value `s`, to pass in a platform type, we could simply do the following:

```
http://{your-GAE-base-url}.appspot.com/videoGameScrapeServlet?type=Xbox
```

Then, on the servlet side do:

```
public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws IOException, ServletException {
    String type = request.getParameter("type");
    games = new ArrayList<VideoGame>();
    String message = "Success";
    try {
        // GRAB GAMES FROM SPECIFIC PLATFORM
        games.addAll(VideoGameScraper.getVideoGamesByConsole(type));
    } catch (Exception e) {
        e.printStackTrace();
        message = "Failed";
    }

    // ADD GAMES TO JDO DATABASE
    VideoGameJDOWrapper.batchInsertGames(games);

    // WRITE A RESPONSE BACK TO ORIGINAL HTTP REQUESTER
    response.setContentType("text/html");
    response.setHeader("Cache-Control", "no-cache");
    response.getWriter().write(message);
}
```

Pretty simple, right? You just have to make sure that the key used in the URL matches the parameter you request within the servlet class. Now, the last and final step for getting this all hooked together is defining the URL path in your GAE project—namely, making sure your GAE project knows that the URL pattern actually points to this class you just wrote. This can be found in your GAE project's `/war/WEB-INF/` directory, specifically in the `web.xml` file. There you'll need to add the following. To make sure that the servlet name and class path matches the given URL pattern:

```
<?xml version="1.0" encoding="utf-8"?>

<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
  <servlet>
    <servlet-name>videoGameScrapeServlet</servlet-name>
    <servlet-class>app.httpservlets.VideoGameScrapeServlet</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>videoGameScrapeServlet</servlet-name>
    <url-pattern>/videoGameScrapeServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

At this point, we have our scraper, we have our JDO database, and we even have our first servlet all hooked up and ready to go. The last part is scheduling your scraper to run periodically; that way, your database has the latest and most up-to-date data, without you having to sit in front of your computer every day and manually call your scraper. In this next section, we'll see how we can use CRON jobs to accomplish just this.

Scheduling CRON jobs

First, let's define what a CRON job is. The term **cron** originally referred to a time-based job scheduler in Unix that allowed you to schedule jobs/scripts to be run periodically at specific times. The same concept can be applied to web requests, and in our case, the goal is to run our web scraper and update the data in our database periodically and without our interference. Another reason why GAE is so convenient to use is because of how easy the platform makes scheduling CRON jobs. To do so, we simply need to create a `cron.xml` file in the `/war/WEB-INF/` directory of our GAE project. In this XML file, we add the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>

  <cron>
    <url>/videoGameScrapeServlet</url>
    <description>Scrape video games from Blockbuster</description>
    <schedule>every day 00:50</schedule>
    <timezone>America/Los_Angeles</timezone>
  </cron>

</cronentries>
```

This is pretty self explanatory. First, we define root tags named `<cronentries>` and within these, we can insert any number of `<cron>` tags—each one denoting a scheduled process. In these `<cron>` tags, we need to tell the scheduler what the URL that we want to hit is (this will be relative to the root URL, of course), as well as the schedule itself (in our case, it's everyday at 12:50 A.M.). Other optional tags are a description tag, a time-zone tag, and/or a target tag that allows you to specify which version of your GAE project to invoke the specified URL.

Now, in my case, I asked the scheduler to run the job every day at 12:50 A.M. (PST), but examples of other schedule formats are as follows:

```
every 12 hours
every 5 minutes from 10:00 to 14:00
2nd,third mon,wed,thu of march 17:00
every monday 09:00
1st monday of sep,oct,nov 17:00
every day 00:00
```

I won't go into the exact syntax of the scheduler tags, but you can see that it's pretty intuitive. However, for those of you who would like to learn more about CRON jobs in GAE or look at some of the less commonly used features, feel free to check out the following URL for a comprehensive look at CRON jobs:

<http://code.google.com/appengine/docs/java/config/cron.html>

But as far as our example goes, what we did previously will suffice and so we'll stop here!

Summary

In this chapter, we yet again covered a lot of ground. We started off the chapter simply looking at various ways to collect data. In some cases, convenient APIs released by other companies are readily available for us to use and query (though one must be careful about legal issues when it comes to storing that data). However, many times we'll find ourselves needing to go out and grab that data ourselves, and this can be done through web scraping.

In the next section, we went through a primer on web scraping – starting with the high-level concepts behind what web scraping is and what steps you need to take to perform it, and ending with the implementation. The example we went through involved scraping Blockbuster's site for the latest video games available for rent, and in the process, we wrote our first XPath expressions and implemented our first HTTP servlet.

While implementing our HTTP servlet, we briefly discussed the two common types of HTTP requests (GET and POST requests) and proceeded to implement an HTTP GET request that would allow us to call our video game scraper class, collect the aggregated `VideoGame` objects, and then insert them into our JDO database using our convenient wrapper class from the previous chapter.

Finally, we ended the chapter by looking at ways in which we could schedule the scraping of Blockbuster's site in order to ensure the latest and most up-to-date data, without having to manually call the scraper ourselves every day. We introduced a special technology known as CRON jobs and implemented one using the GAE platform.

In the next and last chapter, we'll try to bring everything we learned together. More specifically, now that the data collection and insertion parts of our system are up and running, we'll implement a few more servlets that will allow us to make an HTTP GET request and retrieve various types of data. Then, we'll go through the client side of the code, and look at how you can make these GET requests from the Android application and parse the response for the desired data.

10

Bringing it Together

At last, it's time to bring everything together. Earlier in *Chapter 8, Exploring External Databases*, we started our example of writing a Blockbuster games application by creating a new Google App Engine (GAE) project and building up the JDO database. We first defined what our `VideoGame` table should look like, and then we wrote a handful of convenient wrapper methods which would allow us to retrieve, insert, update, and/or delete `VideoGame` data from our backend. Then in *Chapter 9, Collecting and Storing Data*, we looked at various ways in which we could collect data, either by using convenient APIs or by writing scrapers to do the dirty work for us. In our example, a scraper was necessary and so we wrote some code to first clean and structure Blockbuster's game rental page, before finally navigating and parsing the desired data. The last step was simply to reintroduce ourselves to HTTP servlets and look at how we could implement a servlet that, when hit, would scrape and update our database with the latest games.

Now, we'll finish off the application by writing an HTTP servlet that will actually query and return data (as opposed to our earlier example, which simply returned a success or failure message), and, once returned, we'll write some simple XML parsers and list adapters to show you what to do with the data once it's on the mobile side. Then, you'll have a fully functional backend that will periodically scrape and update its own data, a series of HTTP servlets that will allow you to retrieve data independent of the platform, and an Android application that will parse the data and bind it to the UI for the user to see.

Implementing HTTP GET requests

In the last chapter, we briefly went over the difference between a GET and POST request. The next step in our application development is writing a few classes on the GAE server side which will allow us to hit a URL and get back a list of video game objects.

This means we need to override another HTTP servlet which will likely take a parameter that indicates which game platform we're looking for. Intuitively, once we know the platform we're looking for, we recall from earlier that one of our wrapper methods for our JDO database involved querying for all games of a certain platform. Hence, we'll likely need to utilize our JDO wrapper class again.

However, you might also recall that our JDO database returns rows not as strings but as objects, and so we'll need to take the additional step of converting each `VideoGame` object into some kind of readable, formatted string, whether as XML or JSON. With these initial thoughts and intuitions at hand, let's take a look at how you would implement this new GET request:

```
public class GetVideoGames extends HttpServlet {

    // HTTP GET REQUEST SINCE WE'RE REQUESTING FOR DATA
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException, ServletException
    {
        String platform = request.getParameter("type");

        // USE OUR JDO WRAPPER TO QUERY FOR GAMES BY PLATFORM
        List<VideoGame> games =
            VideoGameJDOWrapper.getGamesByType(platform);

        // WRAP GAMES INTO XML FORMAT
        String ret = GamesToXMLConverter.convertGamesToXML(games);

        // SET THE RESPONSE TYPE TO XML
        response.setContentType("text/xml");
        response.setHeader("Cache-Control", "no-cache");

        // WRITE DATA TO RESPONSE
        response.getWriter().write(ret);
    }
}
```

Everything should look familiar and the logic is fairly simple. The only part that's unclear is near the end when I pass in a list of `VideoGame` objects and get back a string. As the name of the class suggests, I wrote a simple class which takes `VideoGame` objects, strips out their fields, and organizes them into well-formatted XML code (again, you could use JSON as well). Let's take a quick look at how I defined my `GamesToXMLConverter` class:

```
public class GamesToXMLConverter {

    public static String convertGamesToXML(List<VideoGame> games) {
        String content = "";
        for (VideoGame g : games) {
            // WRAP EACH GAME IN ITS OWN TAG
            content += convertGameToXml(g);
        }
        // WRAP ALL GAME TAGS TOGETHER INTO ROOT TAG

        String ret = addTag("games", content);
        return ret;
    }

    /**
     * METHOD FOR CONVERTING OBJECT TO XML FORMAT
     *
     * @param g
     *         a video game object
     * @return
     */
    public static String convertGameToXml(VideoGame g) {
        String content = "";
        // ADD TAG FOR NAME
        content += addTag("name", g.getName().replaceAll("&",
            "and"));

        // ADD TAG FOR ID
        content += addTag("id", String.valueOf(g.getId()));

        // ADD TAG FOR IMAGE IF NOT NULL
        if (g.getImgUrl() != null) {
            content += addTag("imageUrl",
                g.getImgUrl().getValue());
        }
    }
}
```

```
        // ADD TAG FOR TYPE
        content += addTag("type",
            VideoGameConsole.convertIntToString(g.getConsoleType()));

        // WRAP ENTIRE GAME IN <game> TAGS
        String ret = addTag("game", content);
        return ret;
    }

    public static String addTag(String tag, String value) {
        return ("<" + tag + ">" + value + "</" + tag + ">");
    }
}
```

And voila – nothing too complicated. Really, you can write your XML/JSON converters in any way you'd like – in fact, if you search hard enough, I'm willing to bet there are convenient libraries out there which are designed to do this for you. However, as is the theme of this book, focus more on the concepts and less on my actual code – the idea is you reach into your JDO database and get back a list of objects and from there you simply need to think of a clean way to write those objects into the `HttpServletResponse` object that is returned.

And again, just like with our previous HTTP servlet, in order for our GAE project to recognize this as a valid servlet, we need to define it as one in the `/war/WEB-INF/web.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>

<servlet>
    <servlet-name>getVideoGames</servlet-name>
    <servlet-class>app.requests.GetVideoGames</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>getVideoGames</servlet-name>
    <url-pattern>/getVideoGames</url-pattern>
</servlet-mapping>
```

And once we have our name and URL pattern defined, we simply deploy the project and hit the following URL:

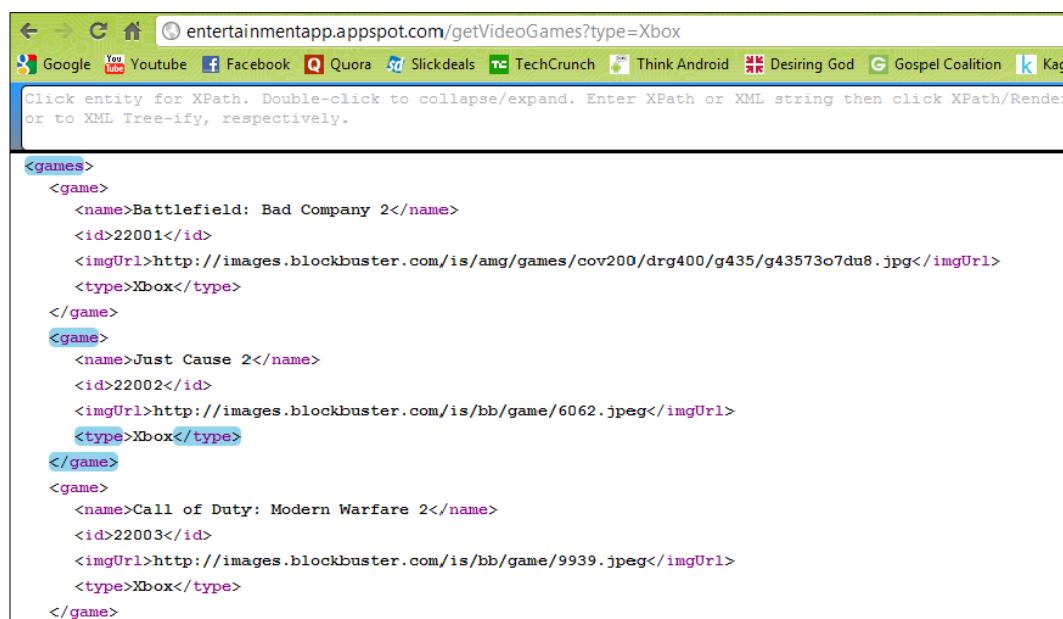
```
http://{your-project-name}.appspot.com/getVideoGames?type={type}
```

And we're done. For those following along, I invite you to check it out and see if you get a nicely formatted list of data. Otherwise, feel free to check out the following links to see my results:

<http://entertainmentapp.appspot.com/getVideoGames?type=Xbox>

<http://entertainmentapp.appspot.com/getVideoGames?type=Ps3>

The following is a screenshot for those reading this on the go:



And now, let's move it back to the Android side and see how we would both make the request and then handle/parse the result.

Back to Android: parsing responses

Now that we have our backend completely finished, all that remains is implementing these HTTP requests from the Android, parsing out the data, and then binding this data to the UI once it's been retrieved (though this will likely be a rehash of *Chapter 6, Binding to the UI*).

To start, you'll need to build an HTTP client which will allow you to make GET/POST requests. What this HTTP client essentially does is it acts as a vehicle for which you can make various HTTP requests. The HTTP client requires that you set some HTTP parameters for how the request should be made. Then, based on those parameters, the client knows how to handle each request accordingly. For instance, one such parameter is telling the HTTP client how to handle HTTP versus HTTPS requests (that is, requests made through an unsecured channel versus a secured one). Each channel requires that you specify a different port, so you'll have to define these accordingly in your client. In the following code you can see an HTTP client which is configured for both HTTP and HTTPS requests:

```
public class ConnectionManager {

    public static DefaultHttpClient getClient() {
        DefaultHttpClient ret = null;

        // SET PARAMETERS
        HttpParams params = new BasicHttpParams();
        HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
        HttpProtocolParams.setContentCharset(params, "utf-8");
        params.setBooleanParameter("http.protocol.expect-continue",
            false);

        // REGISTER SCHEMES FOR HTTP AND HTTPS REQUESTS
        SchemeRegistry registry = new SchemeRegistry();
        registry.register(new Scheme("http",
            PlainSocketFactory.getSocketFactory(), 80));
        final SSLSocketFactory sslSocketFactory =
            SSLSocketFactory.getSocketFactory();
        sslSocketFactory.setHostnameVerifier
            (SSLSocketFactory.BROWSER_COMPATIBLE_HOSTNAME_VERIFIER);
        registry.register(new Scheme("https",
            sslSocketFactory, 443));

        ThreadSafeClientConnManager manager =
            new ThreadSafeClientConnManager(params, registry);
        ret = new DefaultHttpClient(manager, params);
        return ret;
    }
}
```

Once you have that, I prefer to build some simple GET/POST wrapper methods, which when passed an HTTP client and a URL will return the result as a string:

```
public class GetMethods {

    /**
     * MAKE AN HTTP GET REQUEST
     *
     * @param mUrl
     *         the url of the request you're making
     * @param httpClient
     *         a configured http client
     * @return
     */
    public static String doGetWithResponse(String mUrl,
DefaultHttpClient httpClient) {
        String ret = null;
        HttpResponse response = null;

        // INITIATE THE GET METHOD WITH THE DESIRED URL
        HttpGet getMethod = new HttpGet(mUrl);
        try {
            // USE YOUR HTTP CLIENT TO EXECUTE THE METHOD
            response = httpClient.execute(getMethod);
            System.out.println("STATUS CODE: " +
String.valueOf(response.getStatusLine().
getStatusCode()));
            if (null != response) {
                // CONVERT HTTP RESPONSE TO STRING
                ret = getResponseBody(response);
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }

        return ret;
    }

    public static String getResponseBody(HttpResponse response) {
        String response_text = null;
        HttpEntity entity = null;
        try {
            // GET THE MESSAGE BODY OF THE RESPONSE
            entity = response.getEntity();
            if (entity == null) { throw new
IllegalArgumentException("HTTP entity may not be null"); }
        }
    }
}
```



```
// IF NOT NULL GET CONTENT AS STREAM
InputStream instream = entity.getContent();
if (instream == null) { return ""; }

// CHECK FOR LENGTH
if (entity.getContentLength() > Integer.MAX_VALUE)
{ throw new IllegalArgumentException(
  "HTTP entity too large to be buffered in memory"); }

// GET THE CHARACTER SET OF THE RESPONSE
String charset = null;
if (entity.getContentType() != null) {
  HeaderElement values[] = entity.getContentType().
    getElements();
  if (values.length > 0) {
    NameValuePair param = values[0].
      getParameterByName("charset");
    if (param != null) {
      charset = param.getValue();
    }
  }
}
if (charset == null) {
  charset = HTTP.DEFAULT_CONTENT_CHARSET;
}

// ONCE CHARSET IS OBTAINED - READ FROM STREAM
Reader reader = new InputStreamReader(instream, charset);
StringBuilder buffer = new StringBuilder();
try {
  // USE A BUFFER TO READ FROM STREAM
  char[] tmp = new char[2048];
  int l;
  while ((l = reader.read(tmp)) != -1) {
    buffer.append(tmp, 0, l);
  }
} finally {
  reader.close();
}
// CONVERT BUFFER TO STRING
response_text = buffer.toString();
} catch (Exception e) {
  e.printStackTrace();
}
return response_text;
}
}
```

At first glance this may all seem very intimidating, especially for someone who has never seen any of these technologies or classes. Yes – there are a lot of new classes involved, but none of this is rocket science; in fact, the class names are all pretty intuitive and descriptive and there isn't too much going on beyond that.

In the first method, what we're doing is pretty simple. Java has an `HttpGet` class that's included in the Android SDK as well as in the Java SDK, which then gets instantiated with a URL. Next, we pass this `HttpGet` object into our HTTP client and wait for the response to come back to us. The response will eventually come back as an `HttpResponse` object, and within this object there are descriptive fields that tell you the HTTP status code, the content of the response (this is what we'll need shortly), and so on. The status code is a useful thing to have, as it will tell us whether or not the GET request was successful, and, if not, what error it failed with. With these different error codes at hand, we can then handle each event accordingly – for instance, if the server is down, then we're out of luck and should either tell the user to check back later or potentially direct them to an offline version of your application. On the other hand, if it was just a temporary connection issue, then maybe we'll silently make the request again.

Once we have the response and check that it succeeded, it's time to get the response body! The code for that is in the next section – that is, the `getResponseBody()` method. This method is a little more cumbersome, but hopefully the inline comments help guide you through what's going on. From a high level, essentially what we're doing is grabbing the content body of the `HttpResponse` object known in this case as the entity. However, the entity is a separate object which contains numerous descriptive fields, but what we're actually interested in is the string representation of the `HttpEntity` object. Hence, from the `HttpEntity` we request an `InputStream`, which will allow us to utilize a `StringBuilder` object and stream the characters of the content body line by line. Now, the rest of the code in between is simply a series of checks to make sure that there's actually a message to be buffered, and, if so, that it's not too large for our buffer to handle (that is, it doesn't exceed the maximum size of a string). Lastly, we just need to retrieve the content body's character set so that our `InputStreamReader` will know which character set to use when converting the message into characters.

Now, here's how we'll use the two previous classes to actually make the GET request from the Android client side:

```
public class GetVideoGamesAndroid {  
  
    private static String URL_BASE =  
        "http://entertainmentapp.appspot.com";  
  
    private static String REQUEST_BASE = "/getVideoGames?type=";
```

```
// THIS RETRIEVES THE HTTP CLIENT CONFIGURED ABOVE
private static DefaultHttpClient httpClient =
    ConnectionManager.getClient();

// PASS IN THE PLATFORM YOU WANT I.E. XBOX, PS3, ETC
public static List<VideoGame> getGamesByType(String type) {
    // CONSTRUCT GET REQUEST URL
    String url = URL_BASE + REQUEST_BASE + type;

    // XML RESPONSE AS A STRING GETS RETURNED
    String response = GetMethods.doGetWithResponse(url,
        httpClient);

    // RUN THROUGH SIMPLE XML PARSER
    List<VideoGame> games =
        ObjectParsers.parseGameResponse(response);
    return games;
}
}
```

At this point you'll notice that the meat of what's happening is indeed in our `GetMethods` class, and that once this class has been implemented, making GET requests becomes quite simple: one only needs the URL. So then what does the XML parser look like in this case? Well, you can implement it in any number of ways, depending on how complicated the XML is and/or how familiar you are with various XML document parsers. For extremely simple XML (that is, documents with just a single layer of nodes), sometimes using simple REGEX commands will do the trick. In more complex XML, sometimes it helps to use Java's built-in `SAXParser` classes or to even use our buddy `HtmlCleaner`. Note that in many cases the data returned might also be in JSON format, in which case you would need to write some simple JSON parsers that take the various key-value pairs and reconstruct the `VideoGame` objects on the mobile side.

Because of all these previous dependencies, I'll leave the actual implementation of the `parseGameResponse()` method to you guys – the goal is clear and if you need a reminder of what the data looks like, just refer back to the first image of this chapter. Now you just need to parse it, which should be a relatively simple exercise. One last thing I'll mention is that typically these HTTP requests can take some time (at least a couple of seconds, sometimes upwards of 10-20 depending on how much work is being done on the server). Because of how the Android OS will throw an "Application Not Responding" (ANR) error if the main UI thread gets held up for too long (5-10 seconds depending on the condition), I would highly recommend making all HTTP requests on separate threads. You can do this the traditional way using `Runnable` and `Handler` classes, but Android also provides you with nice wrapper classes like the `AsyncTask` class. I'd also encourage you to read this post made by our friends at Google for more on designing responsive applications:

<http://developer.android.com/guide/practices/design/responsiveness.html>

And so now, we've made our GET request, we've parsed the data, and we have a nice list of `VideoGame` objects on the mobile side which are duplicates of the `VideoGame` objects that came from our server. The only thing left to do is use one of our `ListAdapters` which we saw earlier in the book and bind it to the UI!

Final steps: binding to the UI (again)

It's time for the last and final step – binding our data to the user interface. This section should look very familiar for those who have gone through the entire book, so I'll try to be brief but complete.

In the previous sections, we essentially hooked all the network requests together, both on the application side as well as on the server side, so that now we should be able to seamlessly make GET requests from any mobile application. We also looked at ways in which we could parse the resulting response (again, this was left as an exercise, as the response could come back in any number of ways) and convert the data from string form back into `VideoGame` object form.

So now let's think back to *Chapter 6, Binding to the UI*. In that chapter, we looked at two subclasses of `ListAdapters` – the `BaseAdapter` and the `CursorAdapter`. As you'll recall, the `CursorAdapter` is used when our data is stored into a `SQLite` database. The subsequent query into our `SQLite` database is returned in the form of a `Cursor` object which then gets wrapped by the `CursorAdapter` class. In our `VideoGame` example, we currently have a list of objects, not a `Cursor`. That's not to say that we couldn't store our results into a `SQLite` database, effectively making a cache (remember these?) on our application side and then issuing a query into our cache to get back a `Cursor`. But, for simplicity, let's stick with our list of `VideoGame` objects and simply use a `BaseAdapter` which is designed especially for such lists. The code for it might look like the following:

```
public class VideoGameBaseAdpater extends BaseAdapter {

    // REMEMBER CONTEXT SO THAT CAN BE USED TO INFLATE VIEWS
    private LayoutInflater mInflater;

    // LIST OF VIDEO GAMES
    private List<VideoGame> mItems = new ArrayList<VideoGame>();

    public VideoGameBaseAdpater(Context context,
        List<VideoGame> items) {
        // HERE WE CACHE THE INFLATOR FOR EFFICIENCY
        mInflater = LayoutInflater.from(context);
        mItems = items;
    }

    public int getCount() {
        return mItems.size();
    }

    public Object getItem(int position) {
        return mItems.get(position);
    }

    public long getItemId(int position) {
        return position;
    }

    public View getView(int position, View convertView,
        ViewGroup parent) {
        VideoGameViewHolder holder;
```

```

        // IF NULL THEN NEED TO INSTANTIATE IT BY INFLATING IT
        if (convertView == null) {
            convertView = mInflater.inflate(R.layout.list_entry,
                null);

            holder = new VideoGameViewHolder();
            holder.name_entry = (TextView) convertView.findViewById(
                R.id.name_entry);
            holder.type_entry = (TextView) convertView.findViewById(
                R.id.number_type_entry);

            convertView.setTag(holder);
        } else {
            // GET VIEW HOLDER BACK FOR FAST ACCESS TO FIELDS
            holder = (VideoGameViewHolder) convertView.getTag();
        }

        // EFFICIENTLY BIND DATA WITH HOLDER
        VideoGame v = mItems.get(position);
        holder.name_entry.setText(v.getName());

        String type = VideoGameConsole.convertIntToString
            (v.getConsoleType());
        holder.type_entry.setText(type);

        return convertView;
    }

    static class VideoGameViewHolder {
        TextView name_entry;

        TextView type_entry;
    }
}

```

So just like how in *Chapter 6, Binding to the UI*, we implemented a custom `BaseAdpater` that created a list of `Contact` objects – in this case, we're doing something extremely similar but for our `VideoGame` objects! Notice here that my `VideoGameViewHolder` only displays the name of the game and the type of the game and that I'm not doing anything with the image URL. Again, one could easily incorporate this into each row through using an `ImageView`, but that would require converting a URL into a `Bitmap` object – something that's not difficult to do but unnecessary in our case; you get the idea by now.

Now that this is done, we simply need to create an Activity which makes the GET request, takes the resulting list of `VideoGames`, and sets them as its `ListAdapter` by using the custom `VideoGameBaseAdapter`. The code for this is extremely simple:

```
public class VideoGameBaseAdapterActivity extends ListActivity {

    private List<VideoGame> games;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list);

        // MAKE GET REQUEST TO RETRIEVE GAMES
        games = GetVideoGamesAndroid.getGamesByType
            (VideoGameConsole.XBOX);

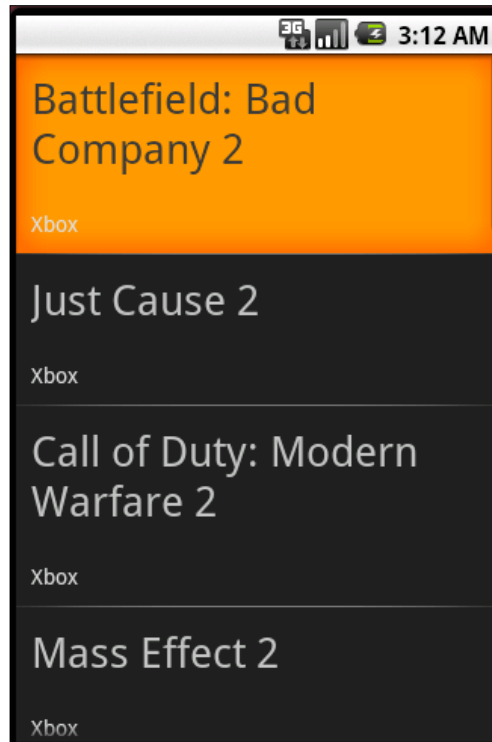
        // USE VIDEO GAME ADAPTER
        VideoGameBaseAdpater vAdapter = new VideoGameBaseAdpater(this,
            games);

        // SET THIS ADAPTER AS YOUR LIST ACTIVITY'S ADAPTER
        this.setAdapter(vAdapter);
    }

    @Override
    protected void onItemClick(ListView l, View v, int position,
        long id) {
        super.onItemClick(l, v, position, id);
        VideoGame vg = games.get(position);

        String name = vg.getName();
        System.out.println("CLICKED ON " + name);
    }
}
```

Once done, our end result looks like the following:



And voila! Pat yourself on the back, as we've now just finished our first full-scale data-centric application! Now not only do we have a fully functional backend equipped with its own set of HTTP requests, but we've also built the beginning of a promising Android application that can make HTTP requests to this backend, obtain the results, and display them in a simple list.

Summary

So, we've reached the end but before we part, let's take a look from start to finish at all the incredible things we've learned and covered. We started this book by looking at various local storage methods on Android – methods that were extremely light-weight and efficient, as well as methods like the SQLite database, which were more complex but at the same time much more powerful.

We then took a deeper look at the SQLite database – likely the most common form of local data storage that you'll encounter in your Android application development careers, before moving onto SQL queries in *Chapter 3, SQLite Queries*. Next, we learned about ways in which we could expose our SQLite databases to external applications through wrapping them in content providers. Then we took a look at the most popular content provider on the Android OS – the Contacts content provider, and implemented some common queries that one might encounter.

Once we had completely mastered local storage methods, we moved on to actually binding these local data sources to the user interface through various `ListAdapter` classes. It was in this chapter that we saw implementations and use cases of both the `CursorAdapter` as well as the `BaseAdapter`.

From there we moved onto a more holistic look at data-centric application design and programming. We talked about practical ways in which we could use all the various forms of local data storage, and also introduced the notion of a cache as one extremely practical use case for a SQLite database. This naturally transitioned us into considering external databases, as caches typically go hand in hand with web requests and web programming.

It was with external databases that we ended our book. We discussed different kinds of external databases that we could use and decided to stick with Google App Engine (GAE) for our sample implementation. It was with GAE that we implemented a fully-functional JDO database (all done in the cloud), at which point we also implemented a series of HTTP servlets that would allow us to make HTTP GET and POST requests. And finally, we ended the book by implementing the code for the mobile side of our application – bringing us full circle and back to Android.

It's my hope that through all this, we can better see how databases, both local and external, fit into the grand scheme of developing powerful, data-centric Android applications. Best of luck and happy developing.

Index

Symbols

<cronentries> tag 175
<cron> tags 175
<friends> tag 158
.mode MODE command 41
.output FILENAME command 41
.tables command 41

A

access speed 17
Activity class
 about 109
 ListActivity 110
addURI() 81
advanced SQLite schemas
 creating 27-30
AGGREGATION_MODE_DEFAULT 97
AGGREGATION_MODE_DISABLED 97
AGGREGATION_MODE_SUSPENDED 97
Amazon's Elastic Compute Cloud. *See* EC2
Amazon's Relational Database Service. *See* RDS
Amazon's Web Services. *See* AWS
AND/OR operators 51
Android
 internal storage methods 13
 need for 7
Android application's backend
 designing 132-134
Android Debug Bridge (adb) 40
Android manifest 78
Apache Tomcat server 141
application design
 steps 137, 138

ArrayAdapter 126
AsyncTask class 187
atomicity 104
AUTOINCREMENT property 21
AVG() aggregate function 61
AWS 143

B

BaseAdapter
 about 118
 comparing, with CursorAdapters 125, 126
 example 124, 125
BaseAdapter class 119
batchInsertGames() method 173
bindView() method 114
buildUnionQuery() method 44

C

cache
 about 134-136
 working 135
CitizensTable class 74
CitizenTable class 88
clean() method 165
clear() method 9
Comma Separated Values. *See* CSV
commit() method 9
ContactBaseAdpater
 implementing 122, 123
ContactEntry class 118
Contacts
 modifying 102-106
 querying 98-102
Contacts ContentProvider
 structure 95-98

- ContactViewHolder class** 121, 122
- ContentProvider**
 - about 73-78
 - delete() method, implementing 82-86
 - getType() method, implementing 86-89
 - insert() method, implementing 89
 - interacting 90-92
 - practical use cases 92, 93
 - query() method, implementing 79-82
 - update() methods, implementing 82-86
- ContentProvider class** 73, 77, 95
- ContentResolver class** 89
- ContentValues class** 23, 104
- COUNT() aggregate function** 61
- COUNT() function** 59
- cron** 174
- CRON jobs**
 - scheduling 174, 175
- cron.xml file** 174
- CSV** 130
- CursorAdapter class** 114, 119
- CursorAdapters**
 - comparing, with BaseAdapter 125, 126
- custom BaseAdapter** 125
- custom CursorAdapters** 117

D

- data**
 - binding, to UI 187-191
- database management system.** *See* DBMS
- databases**
 - as cache 134-136
- data-centric applications** 136
- data collection**
 - ways 157, 158
- data parsing** 181-185
 - GET request, making from Android client side 185, 186
- data, retrieving**
 - ListViews 109
 - SimpleCursorAdapters 109
- DBMS** 142
- delete() method** 37, 82, 85
- DISTINCT clause** 52, 53
- div tag** 166
- doGet() method** 171

E

- EC2** 143
- entities** 144
- execSQL() method** 24
- Extensible Markup Language.** *See* XML
- external databases**
 - about 142, 143
 - Apache Tomcat 141
 - AWS 143
 - DBMS 142
 - EC2 143
 - GAE 143
 - HTTP servlet 142
 - JDO 143
 - MySQL 141
 - RDS 143
 - types 141
- external storage methods** 16-20

G

- GAE**
 - about 143, 177
 - video game example 145-148
- getBoolean() methods** 9
- get() command** 41
- getExternalStorageState() method** 18
- getInt() method** 24
- get() method** 24
- GetMethods class** 186
- getObjectById() method** 153, 155
- get() query** 43
- getResponseBody() method** 185
- getSharedPreferences() method** 8
- getString() method** 24
- getStudentsByGradeForCourse() method** 35
- getType() method** 88
- getVideoGamesByConsole() method** 173
- getView() method** 119, 121
- getWritableDatabase() method** 148
- Google's App Engine setup** 144
- Google's App Engine.** *See* GAE
- GQL** 144
- grabGamesWithTag() method** 169
- GROUPBY clauses** 57, 58

groupBy group 35
GROUPBY statement 68

H

Handler class 187
HAVING filter 59
having group 35
HAVING parameter 60
HtmlCleaner
 URL 161
HtmlCleaner class 165
HttpGet class 185
HTTP GET requests
 implementing 177-181
HTTP requests
 types, GET request 171
 types, POST request 171
HttpResponse object 185
HttpServlet class 171
HTTP servlets
 about 142
 extending 171
 extending, for GET/POST methods 170-174

I

img tag 166
InputStream 185
insert() method 24
internal storage methods
 accessing 14-16

J

Java Database Connectivity. *See* JDBC
Java Data Object. *See* JDO
JavaScript Object Notation. *See* JSON
JDBC 142
JDO 143, 144
JSON 159

K

key-value pairs 8

L

LayoutInflater class 122
LayoutInflater object 116
LIMIT clause
 about 54
 LIMIT n 54
 LIMIT n, m 54
limit parameter 35
ListActivity class 110
ListAdapter. *See* BaseAdapter
ListAdapter
 CursorAdapter 188
 GET BaseAdapter 188
ListAdapter class 109
list interactions
 handling 123, 124
ListView tag 110
localized database
 external databases 131
 external SD cards 130
 pros 132
 SharedPreferences 130
 SQLite databases 131
 use cases 130
local SQLite databases 129
lookup key 98

M

method
 clear() 9
 commit() 9
 delete() 37
 execSQL() 24
 get() 9, 24
 getExternalStorageState() 18
 getSharedPreferences() 8
 getString() 24
 getStringSet() 13
 getStudentsByGradeForCourse() 35
 insert() 24
 onCreate() 8, 21
 onUpgrade() 22, 30
 openFileOutput() 14, 18
 put() 9

query() 37
read() 16
remove() 9
MIME 88
ModelBase class 146
MODE_MULTI_PROCESS mode 9
MODE_WORD_WRITEABLE mode 9
MODE_WORLD_READABLE mode 9
Multipurpose Internet Mail Extensions.
 See **MIME**
MySQL 141

N

newQuery() method 152
newView() method 114

O

onCreate() method 8, 21, 22, 30, 77, 79, 81
onListItemClick() method 123
onUpdate() method 77
onUpgrade() method 22, 30
openFileOutput() method 18
ORDER BY clauses 55-59
orderBy group 35

P

parseGameResponse() method 187
permissions
 setting 107
PersistenceManager 148-157
practical use cases 92, 93
put() methods 9

Q

queries 148
query() method 37, 44, 53, 64, 77, 79, 81
Quick Search widget 93

R

rawQuery() method 43
RDS 143
regular expressions (REGEX) 159
relational databases 20

remove() method 9
Runnable class 187

S

SAXParser classes 186
SchemaHelper class 30
Secure Digital (SD) 16
SELECT * FROM table_name command 41
SELECT * FROM table_name WHERE col = 'value' command 41
SELECT statements
 about 45-49
 results, validating 47-49
setTables() method 65
SharedPreferences
 about 8
 use cases 10
 using, example 8-10
SharedPreferences class 73
SimpleCursorAdapter 109, 112, 116
SQL
 about 20
 COUNT() function 59
 DISTINCT clause 52
 GROUPBY clauses 57, 58
 LIMIT clause 54
 ORDER BY clauses 55-59
 OR operator 51
SQLite 20
SQLite database
 about 27
 debugging 40-42
 instantiating 20-24
 wrappers 30-40
SQLiteDatabase class 24, 44
SQLiteOpenHelper class 21, 27, 75, 95, 148
SQLite queries
 building, methods 43-45
SQLiteQueryBuilder class 44, 47, 50, 52, 65, 150
SQL language performance
 checking 66-70
startManagingCursor() method 92
storage space 16
Structured Query Language. *See* **SQL**
SUM() aggregate function 61

T

TagNode objects 169

U

UI

data, binding to 187-191

Uniform Resource Identifier. *See* URI

UNION SQL query 47

update() method 82, 86

URI 74

use cases, SharedPreferences

application's state, remembering 12

application update, checking 11

first time visit, checking 10

login username, remembering 12

user's location, caching 12, 13

V

VideoGameBaseAdapter 190

W

web scraping 158-160, 170

web.xml file 174

WHERE filter 49, 86

Wi-Fi only filter 135

wrappers

for SQLite database 30-40

X

XML 158

XPath 161



Thank you for buying Android Database Programming

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.





Android 3.0 Animations: Beginner's Guide

ISBN: 978-1-84951-528-3

Paperback: 304 pages

Bring your Android applications to life with stunning animations

1. The first and only book dedicated to creating animations for Android apps.
2. Covers all of the commonly used animation techniques for Android 3.0 and lower versions.
3. Create stunning animations to give your Android apps a fun and intuitive user experience.
4. A step-by-step guide for learning animation by building fun example applications and games.



Android 3.0 Application Development Cookbook

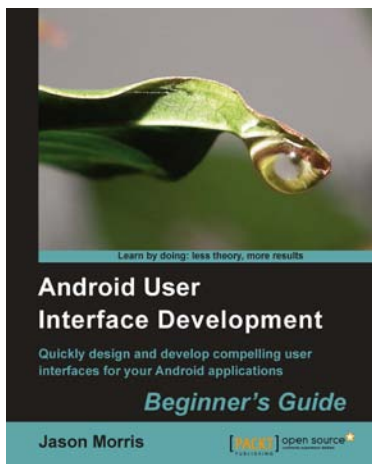
ISBN: 978-1-84951-294-7

Paperback: 272 pages

Over 70 working recipes covering every aspect of Android development

1. Written for Android 3.0 but also applicable to lower versions
2. Quickly develop applications that take advantage of the very latest mobile technologies, including web apps, sensors, and touch screens
3. Part of Packt's Cookbook series: Discover tips and tricks for varied and imaginative uses of the latest Android features

Please check **www.PacktPub.com** for information on our titles



Android User Interface Development: Beginner's Guide

ISBN: 978-1-84951-448-4 Paperback: 304 pages

Quickly design and develop compelling user interfaces for your Android applications

1. Leverage the Android platform's flexibility and power to design impactful user-interfaces
2. Build compelling, user-friendly applications that will look great on any Android device
3. Make your application stand out from the rest with styles and themes
4. A practical Beginner's Guide to take you step-by-step through the process of developing user interfaces to get your applications noticed!



Android Application Testing Guide

ISBN: 978-1-84951-350-0 Paperback: 332 pages

Build intensively tested and bug free Android applications

1. The first and only book that focuses on testing Android applications
2. Step-by-step approach clearly explaining the most efficient testing methodologies
3. Real world examples with practical test cases that you can reuse

Please check www.PacktPub.com for information on our titles

